causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is equal to zero), then execution continues at the "12 aload_0" instruction. If however the performed comparison fails (i.e. if and only if the popped value is not equal to zero), then execution continues at the next instruction in the code stream, which is the "11 return" instruction. This change is particularly significant because it modifies the <init> method to either continue execution of the <init> method (i.e. instructions 12-19) if the returned value of the "isAlreadyLoaded" method was negative (i.e. "false"), or discontinue execution of the <init> method (i.e. the "11 return" instruction causing a return of control to the invoker of this <init> method) if the returned value of the "isAlreadyLoaded" method was positive (i.e. "true").

The method void isAlreadyLoaded(java.lang.Object), part of the InitClient code of Annexure B7, and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the <init> method amongst the machines M1...Mn. The isAlreadyLoaded method of this example communicates with the InitServer code of Annexure B8 executing on a machine X of FIG. 15, by means of sending an "initialization status request" to machine X corresponding to the object being "initialized" (i.e. the object to which this <clinit> method belongs). With reference to FIG. 19 and Annexure B8, machine X receives the "initialization status request" corresponding to the object to which the <clinit> method belongs, and consults a table of initialization states or records to determine the initialization state for the object to which the request corresponds.

If the object corresponding to the initialization status request is not initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object was not already initialized, and update a record entry corresponding to the specified object to indicate the object is now initialized. Alternatively, if the object corresponding to the initialization status request is initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object is already initialized. Corresponding to the determination that the object to which this initialization status request pertains is not

initialized on another machine other than the requesting machine, a reply is generated and sent to the requesting machine indicating that the object is not initialized. Additionally, machine X preferably updates the entry corresponding to the object to which the initialization status request pertained to indicate the object is now initialized. Following a receipt of such a message from machine X indicating that the object is not initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a 'false' value to the previous method frame, which is the executing method frame of the <init> method. Alternatively, following a receipt of a message from machine X indicating that the object is already initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a "true" value to the previous method frame, which is the executing method frame of the <init> method. Following this return operation, the execution of the <init> method frame then resumes as indicated in the code sequence of Annexure B5 at step 006.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of initialization routines or other initialization operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example multiple initialization, or re-initialization operation) does not occur when applying the modified code or procedure.

Annexure B1 is a before-modification excerpt of the disassembled compiled form of the <clinit> method of the example.java application of Annexure B9. Annexure B2 is an after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. Annexure B3 is a before-modification excerpt of the disassembled compiled form of the <init> method of the example.java application of Annexure B9. Annexure B4 is an after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. Annexure B5 is an alternative after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. And Annexure B6 is an alternative after-

modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. The modifications are highlighted in **bold.**

Table X. Annexure B1

```
B1
Method <clinit>
  0 new #2 <Class example>
  3 dup
  4 invokespecial #3 <Method example()>
  7 putstatic #4 <Field example currentExample>
 10 return
```

Table XI. Annexure B2

```
B2
Method <clinit>
  0 invokestatic #3 <Method boolean isAlreadyLoaded()>
  3 ifeq 7
  6 return
  7 new #5 <Class example>
 10 dup
 11 invokespecial #6 <Method example()>
 14 putstatic #7 <Field example example>
 17 return
```

Table XII. Annexure B3

```
B3
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 invokestatic #2 <Method long currentTimeMillis()>
  8 putfield #3 <Field long timestamp>
 11 return
```

Table XIII. Annexure B4

```
B4
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 invokestatic #2 <Method boolean isAlreadyLoaded()>
  7 ifeq 11
 10 return
 11 aload_0
 12 invokestatic #4 <Method long currentTimeMillis()>
 15 putfield #5 <Field long timestamp>
 18 return
```

Table XIV. Annexure B5

```
B5
Method <clinit>
  0 ldc #2 <String "example">
  2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
  5 ifeq 9
  8 return
  9 new #5 <Class example>
 12 dup
 13 invokespecial #6 <Method example()>
 16 putstatic #7 <Field example currentExample>
 19 return
```

Table XV. Annexure B6

```
B6
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
  8 ifeq 12
 11 return
 12 aload_0
 13 invokestatic #4 <Method long currentTimeMillis()>
 16 putfield #5 <Field long timestamp>
 19 return
```

Turning now to FIGS. 20 and 21, the procedure followed to modify class initialisation routines (i.e., the "<clinit>"method) and object initialization routines (i.e. the "<init>" method) is presented. The procedure followed to modify a <clinit> method relating to classes so as to convert from the code fragment of Annexure B1 (See Table X) to the code fragment of Annexure B5 (See Table XIV) is indicated. Similarly, the procedure followed to modify an object initialization <init> method relating to objects so as to convert from the code fragment of Annexure B3 (See Table XII) to the code fragment of Annexure B6 (See Table XV) is indicated.

The initial loading of the application code 50 (an illustrative example in source-code form of which is displayed in Annexure B9, and a corresponding partially disassembled form of which is displayed in Annexure B1 (See also Table X) and Annexure B3 (See also Table XII)) onto the JAVA virtual machine 72 is commenced at step 201, and the code is analysed or scrutinized in order to detect one or more class initialization instructions, code-blocks or methods (i.e. "<clinit>" methods) by carrying out step 202, and/or one or more object initialization instructions, code-blocks, or methods (i.e. "<init>" methods) by carrying out step 212. Once so detected, an <clinit> method is modified by carrying out step 203, and an <init> method is modified by carrying out step 213. One example illustration for a modified class initialisation routine is indicated in Annexure B2 (See also Table XI), and a further illustration of which is indicated in Annexure B5 (See also Table XIV). One example illustration for a modified object initialisation routine is indicated in Annexure B4 (See also Table XIII), and a further illustration of which is indicated in Annexure B6 (See also Table XV). As indicated by step 204 and 214, after the modification is completed the loading procedure is then continued such that the modified application code is loaded into or onto each of the machines instead of the unmodified application code.

Annexure B1 (See also Table X) and Annexure B2 (See also Table XI) are the before (or pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a class initialisation routine (i.e. a "<clinit>" method) respectively. Additionally, a further example of an alternative modified <clinit> method is illustrated in Annexure B5 (See also Table XIV). The modified code that is

added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B1, the "new #2" and "invokespecial #3" instructions of the <clinit> method creates a new object (of the type 'example'), and the following instruction "putstatic #4" writes the reference of this newly created object to the memory location (field) called "currentExample". Thus, without management of coordinated class initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "currentExample" memory location (field) with multiple and different objects corresponding to the multiple executions of the <clinit> method, leading to potentially incoherent or inconsistent memory between and amongst the occurrences of the application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the class initialisation routine (i.e., the <clinit> method). The changes made (highlighted in bold) are the initial instructions that the modified <clinit> method executes. These added instructions determine the initialization status of this particular class by checking if a similar equivalent local class on another machine corresponding to this particular class, has already been initialized and optionally loaded, by calling a routine or procedure to determine the initialization status of the plurality of similar equivalent classes, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method. The "isAlreadyLoaded()" method of InitClient of Annexure B7 of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local classes each on a one of the machines M1, ..., Mn corresponding to the particular class being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, a similar equivalent class.

The initialisation determination procedure or method "isAlreadyLoaded()" of InitClient of Annexure B7 of the DRT 71 can optionally take an argument which represents a unique identifier for this class (See Annexure B5 and Table XIV). For example, the name of the class that is being considered for initialisation, a reference to the class or class-object representing this class being considered for initialization, or a unique number or identifier representing this class across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local classes each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of the plurality of similar equivalent local classes on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple classes at the same time without becoming confused as to which of the multiple classes are already loaded and which are not, by using the unique identifier of each class.

The DRT 71 can determine the initialization status of the class in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local class corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local class has already been initialized, then return a true result at return from the isAlreadyLoaded() method indicating that the local class should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local class should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if one of the plurality of similar equivalent classes on other machines has been initialised.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this class (of the plurality of similar equivalent local classes on the plurality of machines M1...Mn) has not been initialized before on any other machine in the

distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the class initialisation method is to take place or proceed as this is considered the first and original initialization of a class of the plurality of similar equivalent classes on each machine. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this class in the shared record table to true or other value indicating that this class is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and optionally including the current machine, will now return a true value indicating that this class is already initialized. Thus, if isAlreadyLoaded() returns false, the modified class initialisation routine resumes or continues (or otherwise optionally begins or starts) execution.

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this class (of the plurality of similar equivalent local classes each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of classes. In such a case, the class initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions at the start of the <clinit> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <clinit> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this class.

An equivalent procedure for the initialization routines of object (for example "<init>" methods) is illustrated in FIG. 21 where steps 212 and 213 are equivalent to steps 202 and 203 of FIG. 20. This results in the code of Annexure B3 being converted into the code of Annexure B4 (See also Table XIII) or Annexure B6 (See also Table XV).

Annexure B3 (See also Table XII) and Annexure B4 (See also Table XIV) are the before (or pre-modification or unmodified code) and after (or post-modification or

modified code) excerpt of a object initialisation routine (i.e. a "<init>" method) respectively. Additionally, a further example of an alternative modified <init> method is illustrated in Annexure B6 (See also Table XV). The modified code that is added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B4, the "aload_0" and "invokespecial #3" instructions of the <init> method invokes the <init> of the java.lang.Object superclass. Next, the following instructions "aload_0" loads a reference to the 'this' object onto the stack to be one of the arguments to the "8 putfield #3" instruction. Next, the following instruction "invokestatic #2" invokes the method java.lang.System.currentTimeMillis() and returns a long value on the stack. Next the following instruction "putfield #3" writes the long value placed on the stack be the preceding "invokestatic #2" instruction to the memory location (field) called "timestamp" corresponding to the object instance loaded on the stack by the "4 aload_0" instruction. Thus, without management of coordinated object initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "timestamp" memory location (field) with multiple and different values corresponding to the multiple executions of the <init> method, leading to potentially incoherent or inconsistent memory between and amongst the occurrences of application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the object initialisation routine (i.e. the <init> method). The changes made (highlighted in bold) are the initial instructions that the modified <init> method executes. These added instructions determine the initialisation status of this particular object by checking if a similar equivalent local object on another machine corresponding to this particular object, has already been initialized and optionally loaded, by calling a routine or procedure to determine the

initialisation status of the object to be initialised, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method of Annexure B7. The "isAlreadyLoaded()" method of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local objects each on a one of the machines M1, ..., Mn corresponding to the particular object being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, this object.

The initialisation determination procedure or method "isAlreadyLoaded()" of the DRT 71 can optionally take an argument which represents a unique identifier for this object (See Annexure B6 and Table XV). For example, the name of the object that is being considered for initialisation, a reference to the object being considered for initialization, or a unique number or identifier representing this object across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local objects each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of this object in the plurality of similar equivalent local objects on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple objects at the same time without becoming confused as to which of the multiple objects are already loaded and which are not, by using the unique identifier of each object.

The DRT 71 can determine the initialization status of the object in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local object corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local object has already been initialized, then return a true result at return from the isAlreadyLoaded() method indicating that the local object should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local object should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can

consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if this particular object (or any one of the plurality of similar equivalent objects on other machines) has been initialised by one of the requested machines.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this object (of the plurality of similar equivalent local objects on the plurality of machines M1...Mn) has not been initialized before on any other machine in the distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the object initialisation method is to take place or proceed as this is considered the first and original initialization. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this object in the shared record table to true or other value indicating that this object is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and including the current machine, will now return a true value indicating that this object is already initialized. Thus, if isAlreadyLoaded() returns false, the modified object initialisation routine resumes or continues (or otherwise optionally begins or starts) execution..

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this object (of the plurality of similar equivalent local objects each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of objects. In such a case, the object initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions near the start of the <init> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <init> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this object.

A similar modification as used for <clinit> is used for <init>. The application program's <init> method (or methods, as there may be multiple) is or are detected as shown by step 212 and modified as shown by step 213 to behave coherently across the distributed environment.

The disassembled instruction sequence after modification has taken place is set out in Annexure B4 (and an alternative similar arrangement is provided in Annexure B6) and the modified/inserted instructions are highlighted in bold. For the <init> modification, unlike the <clinit> modification, the modifying instructions are often required to be placed after the "invokespecial" instruction, instead of at the very beginning. The reasons for this are driven by the JAVA Virtual Machine specification. Other languages often have similar subtle design nuances.

Given the fundamental concept of testing to determine if initialization has already been carried out on a one of a plurality of similar equivalent classes or object or other asset each on a one of the machines M1...Mn, and if not carrying out the initialization, and if so, not carrying out the initialization; there are several different ways or embodiments in which this coordinated and coherent initialization concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of an initialisation routine, modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other asset or resource) inclusive of the new modified initialization routine(s). Note that there may be one or a plurality of routines corresponding to only one object in the application code, or there may be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the initialization routine(s) that is (are) loaded is binary executable object code. Alternatively, the initialization routine(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified initialisation routine(s), that was sent to it over the computer communications network or other communications link or path by the

master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads asset (such as class or object) inclusive of an (or even one or more) initialization routine in unmodified form on machine M2, and then (for example, machine M2 or each local machine) modifies the class (or object or asset) by deleting the initialization routine in whole or part from the asset (or class or object) and loads by means of a computer communications network or other communications link or path the modified code for the asset with the now modified or deleted initialization routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset initialization routine but a deletion of the initialization routine on all machines except one.

The process of deleting the initialization routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3, ..., Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or

"primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code (including for example, the modified code) for the asset, class or object between machines M1, M2,...,Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more initialization routines, but modifies the routines and then loads the asset (such as class or object) consisting of the now modified routines. Although one machine, such as the master or primary machine may customize or perform a different modification to the initialization routine sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of one or more initialisation routine and all other machines M2, M3, ..., Mn perform a modification to delete the initialization routine of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the initial machine, say M2, can carry out the initial loading of the application code 50, modify it in accordance with this invention, and then generate a class/object loaded and initialised table which lists all or at least all the pertinent classes and/or objects loaded and initialised by machine M2. This

table is then sent or communicated (or at least its contents are sent or communicated) to all other machines (including for example in branched or cascade fashion). Then if a machine, other than M2, needs to load and therefore initialise a class listed in the table, it sends a request to M2 to provide the necessary information, optionally consisting of either the unmodified application code 50 of the class or object to be loaded, or the modified application code of the class or object to be loaded, and optionally a copy of the previously initialised (or optionally and if available, the latest or even the current) values or contents of the previously loaded and initialised class or object on machine M2. An alternative arrangement of this mode may be to send the request for necessary information not to machine M2, but some other, or even more than one of, machine M1, ..., Mn or machine X. Thus the information provided to machine Mn is, in general, different from the initial state loaded and initialise by machine M2.

Under the above circumstances it is preferable and advantageous for each entry in the table to be accompanied by a counter which is incremented on each occasion that a class or object is loaded and initialised on one of the machines M1, ..., Mn. Thus, when data or other content is demanded, both the class or object contents and the count of the corresponding counter, and optionally in addition the modified or unmodified application code, are transferred in response to the demand. This "on demand" mode may somewhat increase the overhead of the execution of this invention for one or more machines M1, ..., Mn, but it also reduces the volume of traffic on the communications network which interconnects the computers and therefore provides an overall advantage.

In a still further arrangement, the machines M1 to Mn, may send some or all load requests to an additional machine X (see for example the embodiment of FIG. 15), which performs the modification to the application code 50 inclusive of an (and possibly a plurality of) initialisation routine(s) via any of the afore mentioned methods, and returns the modified application code inclusive of the now modified initialization routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application code inclusive of the modified routines locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code 50 inclusive of modified initialization

routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

FINALIZATION

Turning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that the platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Furthermore, when there is only a single computer or machine 72, the single machine of FIG. 14 is able to easily keep track of whether the specific objects 50X, 50Y, and/or 50Z are, liable to be required by the application code 50 at a later point of execution of the application code 50. This may typically be done by maintaining a "handle count" or similar count or index for each object and/or class. This count may typically keep track of the number of places or times in the executing application code 50 where reference is made to a specific object (or class). For a handle count (or other count or index based) implementation that increments the handle count (or index) upward when a new reference to the object or class is created or assigned, and decrements the handle count (or index) downward when a reference to the object or class is destroyed or lost, when the object handle count for a specific object reaches zero, there is nowhere in the executing application code 50 which makes reference to the specific object (or class) for which the zero object handle count (or class handle count) pertains. For example, in the JAVA language and virtual machine environment, a "zero object handle count" correlates to the lack of the existence of any references (zero reference count) which point to the specific object. The object is then said to be "finalizable" or exist in a finalizable state. Object handle counts (and handle counters) may be maintained for each object in an analogous manner so that finalizable or non-finalizable state of each particular or specific object may be known.

Class handle counts (and class handle counters) may be maintained for each class in an analogous manner to that for objects so that finalizable or non-finalizable state of each particular or specific class may be known. Furthermore, asset handle counts or indexes and counters may be maintained for each asset in an analogous manner to that for classes and objects so that finalizable or non-finalizable state of each particular or specific asset may be known.

Once this finalizable state has been achieved for an object (or class), the object (or class) can be safely finalized. This finalization may typically include object (or class) deletion, removal, clean-up, reclamation, recycling, finalization or other memory freeing operation because the object (or class) is no longer needed.

Therefore, in light of the availability of these reference, pointer, handle count or other class and object type tracking means, the computer programmer (or other automated or nonautomated program generator or generation means) when writing a program such as the application code 50 using the JAVA language and architecture, need not write any specific code in order to provide for this class or object removal, clean up, deletion, reclamation, recycling, finalization or other memory freeing operation. As there is only a single JAVA virtual machine 72, the single JAVA virtual machine 72 can keep track of the class and object handle counts in a consistent, coherent and coordinated manner, and clean up (or carry out finalization) as necessary in an automated and unobtrusive fashion, and without unwanted behaviour for example erroneous, premature, supernumerary, or re-finalization operation such as may be caused by inconsistent and/or incoherent finalization states or handle counts. In analogous manner, a single generalized virtual machine or machine or runtime system can keep track of the class and object handle counts (or equivalent if the machine does not specifically use "object" and "class" designations) and clean up (or carry out finalization) as necessary in an automated and unobtrusive fashion.

The automated handle counting system described above is used to indicate when an object (or class) of an executing application program 50 is no longer needed and may be 'deleted' (or cleaned up, or finalized, or reclaimed, or recycled, or other otherwise freed). It is to be understood that when implemented in 'non-automated memory management' languages and architectures (such as for example 'non-garbage

collected' programming languages such as C, C++, FORTRAN, COBOL, and machine-CODE languages such as x86, SPARC, PowerPC, or intermediate-code languages), the application program code 50 or programmer (or other automated or non-automated program generator or generation means) may be able to make the determination at what point a specific object (or class) is no longer needed, and consequently may be 'deleted' (or cleaned up, or finalized, or reclaimed, or recycled). Thus, 'deletion' in the context of this invention is to be understood to be inclusive of the deletion (or cleaning up, or finalization, or reclamation, or recycling, or freeing) of objects (or classes) on 'non-automated memory management' languages and architectures corresponding to deletion, finalization, clean up, recycling, or reclamation operations on those 'non-automated memory management' languages and architectures.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2 .... Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines is provided with a modifier 51 (See FIG. 5) and realised or implemented by or in for example the distributed run-time system (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71, or other code modifying means is responsible for modifying the application code 50 so that it may execute clean up or other memory reclamation, recycling, deletion or finalization operations, such as for example finalization methods in the JAVA language and virtual machine environment, in a coordinated, coherent and consistent manner across and between the plurality of individual machines M1, M2, ..., Mn. It follows therefore that in such a computing environment it is necessary to ensure that the local objects and classes on

each of the individual machines is finalized in a consistent fashion (with respect to the others).

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

In particular, whilst the application program code executing on one particular machine (say, for example machine M3) may have no active handle, reference, or pointer to a specific local object or class (i.e. a "zero handle count"), the same application program code executing on another machine (say for example machine M5) may have an active handle, reference, or pointer to the local similar equivalent object or class corresponding to the 'un-referenced' local object or class of machine M3, and therefore this other machine (machine M5) may still need to refer to or use that object or class in future. Thus if the corresponding similar equivalent local object

or class on each machine M3 and M5 were to be finalized (or otherwise cleaned-up by some other memory clean-up operation) in an independent and uncoordinated manner relative to other machine(s), the behaviour of the object and application as a whole is undefined – that is, in the absence of coordinated, coherent, and consistent finalization or memory clean-up operations between machines M1...Mn, conflict, unwanted interactions, or other anomalous behaviour such as permanent inconsistency between local similar equivalent corresponding objects on machine M5 and machine M3 is likely to result. For example, if the local similar equivalent object or class on machine M3 were to be finalized, such as by being deleted, or cleaned up, or reclaimed, or recycled, from machine M3, in an uncoordinated and inconsistent manner with respect to machine M5, then if machine M5 were to perform an operation on or otherwise use the local object or class corresponding to the now finalized similar equivalent local object on machine M3 (such operation being for example, in an environment with a memory updating and propagation means of FIGS. 9, 10, 11, 12, and 13, a write to (or try to write to) the similar equivalent local object on machine M5 or amendment to that particular object's value), then that operation (the change or attempted change in value) could not be performed (propagated from machine M5) throughout all the other machines M1, M2...Mn since at least the machine M3 would not include the relevant similar equivalent corresponding particular object in its local memory, the object and its data, contents and value(s) having been deleted by the prior object clean-up or finalization or reclamation or recycling operation. Therefore, even though one may contemplate machine M5 being able to write to the object (or class) the fact that it has already been finalized on machine M3 means that likely such a write operation is not possible, or at the very least not possible on machine M3.

Additionally, if an object of class on machine M3 were to be marked finalizable and subsequently finalized (such as by being deleted, or cleaned up, or reclaimed, or recycled) whilst the same object on the other machines M1, M2...Mn were not also marked as finalizable, then the execution of the finalization (or deletion, or clean up, or reclamation, or recycling) operation of that object on machine M3 would be premature with respect to coordinated finalization operation between all machines M1, M2...Mn, as machines other than M3 are not yet ready to finalize their

local similar equivalent object corresponding to the particular object now finalized or finalizable by machine M3. Therefore were machine M3 to execute the cleanup or other finalization routine on a given particular object (or class), the cleanup or other finalization routine would preform the clean-up or finalization not just for that local object (or class) on machine M3, but also for all similar equivalent local objects or classes (i.e. corresponding to the particular object or class to be cleaned-up or otherwise finalized) on all other machines as well.

Were such either these circumstance to happen, the behaviour of the equivalent object on the other machines M1, M2...Mn is undefined and likely to result in permanent and irrecoverable inconsistency between machine M3 and machines M1, M2...Mn. Therefore, though machine M3 may independently determine an object (or class) is ready for finalization and proceed to finalize the specified object (or class), machine M5 may not have made the same determination as to the same similar equivalent local object (or class) being ready to be finalized, and therefore inconsistent behaviour will likely result due to the deletion of one of the plurality of similar equivalent objects on one machine (eg, machine M3) but not on the other machine (eg, machine M5) or machines, and the premature execution of the finalization routine of the specified object (or class) by machine M3 and on behalf of all other machines M1,M2...Mn. At the very least operation of machine M5 as well as other machines in such as an above circumstance is unpredictable and would likely lead to inconsistent results, such inconsistency potentially arising for example from, uncoordinated premature execution of the finalization routine and/or deletion of the object on one, or a subset of, machines but not others. Thus, a goal of achieving or providing consistent coordinated finalization operation (or other memory clean-up operation) as required for the simultaneous operation of the same application program code on each of the plurality of machines M1, M2...Mn would not be achieved. Any attempt therefore to maintain identical memory contents with a memory updating and propagation means of FIGS. 9, 10, 11, 12, and 13, or even identical memory contents as to a particular or defined set of classes, objects, values, or other data, for each of the machines M1, M2, ..., Mn, as required for simultaneous operation of the same application program, would not be achieved given conventional schemes.

In order to ensure consistent class and object (or equivalent) finalizable status and finalization or clean up between and amongst machines M1, M2, ..., Mn, the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent a finalization operation or routine (or other memory, data, or code clean up routine, or other similar reclamation, recycling, or deletion operation). In the JAVA language, such program steps may for example comprise or consist of some part of, or all of, a "finalize()" method of an object, and optionally any other code, routine, or method related to a 'finalize()' method, for example by means of a method invocation from the body of the 'finalize()' method to a different method.

This analysis or scrutiny may take place either prior to loading the application program, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application program may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as from source-code or intermediate-code language to machine language), and with the understanding that the term compilation normally involves a change in code or language, for example, from source to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application

code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As a consequence, of the above described analysis or scrutiny, clean up routines are initially looked for, and when found or identified a modifying code is inserted so as to give rise to a modified clean up routine. This modified routine is adapted and written to abort the clean up routine on any specific machine unless the class or object (or in the more general case to be 'asset') to be deleted, cleaned up, reclaimed, recycled, freed, or otherwise finalized is marked for deletion by all other machines. There are several different alternative modes wherein this modification and loading can be carried out.

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device and copying it into memory whilst preparing to begin execution of the application program. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the execution of the java.lang.ClassLoader loadClass (e.g., "java.lang.ClassLoader.loadClass()") method.

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure such as after the operating system has loaded the application code into memory and even started execution, or after the java virtual machine has loaded the application code into the

virtual machine via the "java.lang.ClassLoader.loadClass()" method. In other words, in the case of the JAVA virtual machine, after the execution of "java.lang.ClassLoader.loadclass()" has concluded.

Thus, in one mode, the DRT 71/1 on the loading machine, in this example Java Machine M1 (JVM#1), asks the DRT's 71/2, ..., 71/n of all the other machines M2, ..., Mn if the similar equivalent first object 50X on all machines, say, is utilized, referenced, or in-use (i.e. not marked as finalizable) by any other machine M2, ..., Mn. If the answer to this question is yes (that is, a similar equivalent object is being utilized by another one or more of the machines, and is not marked as finalizable and therefore not liable to be deleted, cleaned up, finalized, reclaimed, recycled, or freed), then the ordinary clean up procedure is turned off, aborted, paused, or otherwise disabled for the similar equivalent first object 50X on machine JVM#1. If the answer is no, (that is the similar equivalent first object 50X on each machine is marked as finalizable on all other machines with a similar equivalent object 50X) then the clean up procedure is operated (or resumed or continued, or commenced) and the first object 50X is deleted not only on machine JVM#1 but on all other machines M2...Mn with a similar equivalent object 50X. Preferably, execution of the clean up routine is allocated to one machine, such as the last machine M1 marking the similar equivalent object or class as finalizable. The execution of the finalization routine corresponding to the determination by all machines that the plurality of similar equivalent objects is finalizable, is to execute only once with respect to all machines M1...Mn, and preferably by only one machine, on behalf of all machines M1....Mn. Corresponding to, and preferably following, the execution of the finalization routine, all machines may then delete, reclaim, recycle, free or otherwise clean-up the memory (and other corresponding system resources) utilized by their local similar equivalent object.

Annexures C1, C2, C3, and C4 (also reproduced in part in Tables XVI, XVII, XVIII, and XIX below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a finalization routine of application program 50 (Annexure C1 and Table XVI), and a post-modification excerpt of the same synchronization routine such as may be used in embodiments of the present invention

having multiple machines (Annexures C2 and C3 and Tables XVII and XVIII). Also the modified code that is added to the finalization routine is highlighted in **bold** text.

Annexure C1 is a before-modification excerpt of the disassembled compiled form of the finalize() method of the example.java application of Annexure C4. Annexure C2 is an after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of FIG. 22. Annexure C3 is an alternative after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of FIG. 22. The modifications are highlighted in **bold.**

Annexure C4 is an excerpt of the source-code of the example.java application used in before/after modification excerpts C1-C3. This example application has a single finalization routine, the finalize() method, which is modified in accordance with this invention by FinalLoader.java of Annexure C7.

Table XVI. Annexure C1- Typical prior art finalization for a single machine

```
Method finalize()
0 getstatic #9 <Field java.io.PrintStream out>
3 ldc #24 <String "Deleted...">
5 invokevirtual #16 <Method void println(java.lang.String)>
8 return
```

Table XVII. Annexure C2 - Finalization For Multiple Machines

```
Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return
```

Table XVIII. Annexure C3 - Finalization For Multiple Machines (Alternative)

```
Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return
```

Table XIX. Annexure C4 - Source-code of the example.java application used in before/after modification excerpts of Annexures C1-C3

```
import java.lang.*;

public class example{

   /** Finalize method. */
   protected void finalize() throws Throwable{

      // "Deleted..." is printed out when this object is garbaged.
      System.out.println("Deleted...");

   }
}
```

It is noted that the compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure C4. In the procedure of Annexure C1 and Table XVI, the procedure name "Method finalize()" of Step 001 is the name of the displayed disassembled output of the finalize method of the compiled application code "example.java". The method name "finalize()" is the name of an object's finalization method in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA finalization method. Overall the method is responsible for disposing of system resources or to perform other cleanup corresponding to the

determination by the garbage collector of a JAVA virtual machine that there are no more references to this object, and the steps the "example.java" code performs are described in turn.

First (Step 002), the JAVA virtual machine instruction "getstatic #9 <Field java.io.PrintStream out>" causes the JAVA virtual machine to retrieve the object reference of the static field indicated by the CONSTANT_Fieldref_info constant_pool item stored in the 2nd index of the classfile structure of the application program containing this example finalize() method and results on a reference to a java.io.PrintStream object in the field to be placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the JAVA virtual machine instruction "ldc #24 <String "Deleted...">" causes the JAVA virtual machine to load the String value "Deleted" onto the stack of the current method frame and results in the String value "Deleted" loaded onto the top of the stack of the current method frame.

Next (Step 004), the JAVA virtual machine instruction "invokevirtual #16 <Method void println(java.lang.String)>" causes the JAVA virtual machine to pop the topmost item off the stack of the current method frame and invoke the "println" method, passing the popped item to the new method frame as its first argument, and results in the "println" method being invoked.

Finally, the JAVA virtual machine instruction "return" (Step 005) causes the JAVA virtual machine to cease executing this finalize() method by returning control to the previous method frame and results in termination of execution of this finalize method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the object handle count in a consistent, coherent and coordinated manner, and in executing the finalize() method containing the println operation is able to ensure that unwanted behaviour (for example premature or supernumerary finalization operation such as execution of the finalize() method of a single 'example.java' object more than once) such as may be caused by inconsistent and/or incoherent finalization states or handle counts, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and

propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the finalization operations of each concurrently executing application program occurrence on each of the one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the goal of consistent, coordinated and coherent finalization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated finalization operation across the plurality of machines, as each machine performs finalization only locally and without any attempt to coordinate their local finalization operation with any other similar finalization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent finalization states or handle counts, and associated finalization operation. Therefore it is the goal of the present invention to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XVIII (Annexure C3), the code has been modified so that it solves the problem of consistent, coordinated finalization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table XVI (Annexure C1). In this modified finalize() method code, an "aload_0" instruction is inserted before the "getstatic #9" instruction in order to be the first instruction of the finalize() method. This causes the JAVA virtual machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame, and results in the object reference of the 'this' object at index 0 being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method boolean isLastReference(java.lang.Object)>" is inserted after the "0 aload_0" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0" instruction is a reference to the object to which this finalize() method belongs) and invokes the "isLastReference" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this "invokestatic" instruction. This change is significant because it modifies the

finalize() method to execute the "isLastReference" method and associated operations, corresponding to the start of execution of the finalize() method, and returns a boolean argument (indicating whether the object corresponding to this finalize() method is the last remaining reference amongst the similar equivalent object on each of the machines M1...Mn) onto the stack of the executing method frame of the finalize() method.

Next, two JAVA virtual machine instructions "ifne 8" and "return" are inserted into the code stream after the "1 invokestatic #3" instruction and before the "getstatic #9" instruction. The first of these two instructions, the "ifne 8" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is not equal to zero), then execution continues at the "8 getstatic #9" instruction. If however the performed comparison fails (i.e. if and only if the popped value is equal to zero), then execution continues at the next instruction in the code stream, which is the "7 return" instruction. This change is particularly significant because it modifies the finalize() method to either continue execution of the finalize() method (i.e. instructions 8-16) if the returned value of the "isLastReference" method was positive (i.e. "true"), or discontinue execution of the finalize() method (i.e. the "7 return" instruction causing a return of control to the invoker of this finalize() method) if the returned value of the "isLastReference" method was negative (i.e. "false").

The method void isLastReference(java.lang.Object), part of the FinalClient code of Annexure C5 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the finalize() method amongst the machines M1...Mn. The isLastReference method of this example communicates with the InitServer code of Annexure C6 executing on a machine X of FIG. 15, by means of sending an "clean-up status request" to machine X corresponding to the object being "finalized" (i.e. the object to which this finalize() method belongs). With reference to FIG. 25 and Annexure C6, machine X receives the "clean-up status request" corresponding to the object to which the finalize() method belongs, and consults a table of clean-up counts

or finalization states to determine the clean-up count or finalization state for the object to which the request corresponds.

If the plurality of similar equivalent objects one each one of the plurality of machines M1...Mn corresponding to the clean-up status request is marked for clean-up on all other machines than the requesting machine (i.e. n-1 machines), then machine X will send a response indicating that the plurality of similar equivalent objects are marked for clean-up on all other machines, and optionally update a record entry corresponding to the specified similar equivalent objects to indicate the similar equivalent objects as now cleaned up. Alternatively, if the plurality of the similar equivalent objects corresponding to the clean-up status request is not marked for clean-up on all other machines than the requesting machine (i.e. less than n-1 machines), then machine X will send a response indicating that the plurality of similar equivalent objects is not marked for cleanup on all other machines, and increment the "marked for clean-up counter" record (or other similar finalization record means) corresponding to the specified object, to record that the requesting machine has marked its one of the plurality of similar equivalent objects to be cleaned-up. Corresponding to the determination that the plurality of similar equivalent objects to which this clean-up status request pertains is marked for clean-up on all other machines than the requesting machine, a reply is generated and sent to the requesting machine indicating that the plurality of similar equivalent objects is marked for clean-up on all other machines than the requesting machine. Additionally, and optionally, machine X may update the entry corresponding to the object to which the clean-up status request pertained to indicate the plurality of similar equivalent objects as now "cleaned-up". Following a receipt of such a message from machine X indicating that the plurality of similar equivalent objects is marked for clean-up on all other machines, the isLastReference() method and operations terminate execution and return a 'true' value to the previous method frame, which is the executing method frame of the finalize() method. Alternatively, following a receipt of a message from machine X indicating that the plurality of similar equivalent objects is not marked for clean-up on all other machines, the isLastReference() method and operations terminate execution and return "false" value to the previous method frame, which is the executing method frame of the finalize() method. Following this return operation,

the execution of the finalize() method frame then resumes as indicated in the code sequence of Annexure C3.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of finalization routines or other clean-up operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example erroneous, premature, multiple finalization, or re-finalization operation) does not occur when applying the modified code or procedure.

It may be observed that the code in Annexure C2 and Table XVII is an alternative but lesser preferred form of the code in Annexure C3. It is essentially functionally equivalent to the code and approach in Annexure C3.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2, ..., Mn are as before and run the same application code 50 (or codes) on all machines M1, M2, ..., Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the clean up of structures, assets and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2, ..., Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After loading 161 has been commenced, the instructions to be executed are considered in sequence and all clean up routines are detected as indicated in step 162. In the JAVA language these are the finalization routines or finalize method (e.g., "finalize( )"). Other languages use different terms.

Where a clean up routine is detected, it is modified at step 163 in order to perform consistent, coordinated, and coherent clean up or finalization across and between the plurality of machines M1, M2...Mn, typically by inserting further instructions into the clean up routine to, for example, determine if the object (or class or other asset) containing this finalization routine is marked as finalizable across all similar equivalent local objects on all other machines, and if so performing finalization by resuming the execution of the finalization routine, or if not then aborting the execution of the finalization routine, or postponing or pausing the execution of the finalization routine until such a time as all other machines have marked their similar equivalent local objects as finalizable. Alternatively, the modifying instructions could be inserted prior to the routine. Once the modification has been completed the loading procedure continues by loading modified application code in place of the unmodified application code, as indicated in step 164. Altogether, the finalization routine is to be executed only once, and preferably by only one machine, on behalf of all machines M1...Mn corresponding to the determination by all machines M1...Mn that the particular object is finalizable.

FIG. 17 illustrates a particular form of modification. Firstly, the structures, assets or resources (in JAVA termed classes or objects) 50A, 50X...50Y which are possible candidates to be cleaned up, are allocated a name or tag (for example a global name or tag), or have already been allocated a global name or tag, which can be used to identify corresponding similar equivalent local structures, assets, or resources (such as classes and objects in JAVA) globally on each of the machines M1, M2...Mn, as indicated by step 172. This preferably happens when the classes or objects are originally initialized. This is most conveniently done via a table maintained by server machine X. This table also includes the "clean up status" of the class or object (or other asset). It will be understood that this table or other data structure may store only the clean up status, or it may store other status or information as well. In one embodiment, this table also includes a counter which stores a machine asset deletion count value identifying the number of machines (and optionally the identity of the machines although this is not required) which have marked this particular object, class, or other asset for deletion. In one embodiment, the count value is incremented until the count value equals the number of machines. Thus a total machine asset

deletion count value of less than (n-1), where n is the total number of machines in Mn indicates a "do not clean up" status for the object, class, or other asset as a network (or machine constellation) whole, because the machine asset deletion count of less than n-1 means that one or more machines have yet to mark their similar equivalent local object (or class or other asset) as finalizable and that object cannot be cleaned up as unwanted or other anomalous behaviour may result. Stated differently, and by way of example but not limitation, if there are six machines and the asset deletion count is less than five then it means that not all the other machines have attempted to finalize this object (i.e., not yet marked this object as finalizable), and therefore the object can't be finalised. If however the asset deletion count is five, then it means that there is only one machine that has yet to attempt to finalize this object (i.e., mark this object as finalizable) and therefore that last machine yet to mark the object as finalizable must be the current machine attempting to finalize the object (i.e., marking the object as finalizable and consequently consulting the finalization table as to finalization status of this object on all other machines). In the configuration of six machines, the count value of n-1=5 means that five machines must have previously marked the object for deletion and the sixth machine to mark this object for deletion is the machine that actually executes the full finalization routine.

As indicated in FIG. 17, if the global name or identifier is not marked for cleanup or deletion or other finalization on all other machines (i.e., all except on the machine proposing to carry out the clean up or deletion routine) then this means that the proposed clean up or finalization routine of the object or class (or other asset) should be aborted, stopped, suspend, paused, postponed, or cancelled prior to its initiation or if already initiated then to its completion if it has already begun execution, since the object or class is still required by one or more of the machines M1, M2...Mn, as indicated by step 175.

In one embodiment, the clean up or finalization routine is stopped from initiating or beginning execution; however, if some implementations it is difficult or practically impossible to stop the clean up or finalization routine from initiating or beginning execution. Therefore, in an alternative embodiment, the execution of the finalization routine that has already started is aborted such that it does not complete or does not complete in its normal manner. This alternative abortion is understood to

include an actual abortion, or a suspend, or postpone, or pause of the execution of a finalization routine that has started to execute (regardless of the stage of execution before completion) and therefore to make sure that the finalization routine does not get the chance to execute to completion to clean up the object (or class or other asset), and therefore the object (or class or other asset) remains "uncleaned" (i.e., "unfinalised", or "not deleted").

However or alternatively, if the global name or other unique number or identifier for a plurality of similar equivalent local objects each on of the plurality of machines M1, M2...Mn is marked for deletion on all other machines, this means that no other machine requires the class or object (or other asset) corresponding to the global name or other unique number or identifier. As a consequence clean up routine and operation, or optionally the regular or conventional ordinary clean up routine and operation, indicated in step 176 can be, and should be, carried out.

FIG. 18 shows the enquiry made by the machine proposing to execute a clean up routine (one of M1, M2...Mn) to the server machine X. The operation of this proposing machine is temporarily interrupted, as shown in step 181 and 182, and corresponding to step 173 of FIG. 17. In step 181 the proposing machine sends an enquiry message to machine X to request the clean-up or finalization status of the object (or class or other asset) to be cleaned-up. Next, the proposing machine awaits a reply from machine X corresponding to the enquiry message sent by the proposing machine at step 181, indicated by step 182.

FIG. 25 shows the activity carried out by machine X in response to such a finalization or clean up status enquiry of step 181 in FIG. 18. The finalization or clean up status is determined as seen in step 192 which determines if the object (or class or other asset) corresponding to the clean-up status request of global name, as received at step 191 (191A), is marked for deletion on all other machines other than the enquiring machine 181 from which the clean-up status request of step 191 originates. The singular term object or class as used in this document (or the equivalent term of asset, or resource used in step 192 (192A) and other Figures) are to be understood to be inclusive of all similar equivalent objects (or classes, or assets, or resources) corresponding to the same global name on each of the plurality of machines M1, M2, ..., Mn. If the step 193 (193A) determination is made that

determines that the global named resource is not marked ("No") for deletion on (n-1) machines (i.e. is utilized elsewhere), then a response to that effect is sent to the enquiring machine 194 (194A) but the "marked for deletion" counter is incremented by one (1), as shown by step 197 (197A). Similarly, if the answer to this determination is marked for deletion ("Yes") indicating that the global named resource is marked for deletion on all other machines other than the waiting enquiring machine 182 then a corresponding reply is sent to the waiting enquiring machine 182 from which the clean-up status request of step 191 originated as indicated by step 195 (195A). The waiting enquiring machine 182 is then able to respond accordingly, such as for example by: (i) aborting (or pausing, or postponing) execution of the finalization routine when the reply from machine X of step 182 indicated that the similar equivalent local objects on the plurality of machines M1, M2, ..., Mn corresponding to the global name of the object proposed to be finalized of step 172 is still utilized elsewhere (i.e., not marked for deletion on all other machines other than the machine proposing to carry out finalization); or (ii) by continuing (or resuming, or starting) execution of the finalization routine when the reply from machine X of step 182 indicated that the similar equivalent local objects on the plurality of machines M1, M2...Mn corresponding to the global name of the object proposed to be finalized of step 172 are not utilized elsewhere (i.e., marked for deletion on all other machines other than the machine proposing to carry out finalization). As indicated by broken lines in FIG. 25, preferably in addition to the "yes" response shown in step 195, the shared table or cleaned-up statuses stored or maintained on machine X is updated so that the status of the globally named asset is changed to "cleaned up" as indicated by step 196.

Reference is made to the accompanying Annexure C in which: Annexure C1 is a typical code fragment from an unmodified finalize routine, Annexure C2 is an equivalent in respect of a modified finalize routine, and Annexure C3 is an alternative equivalent in respect of a modified finalize routine.

Annexures C1 and C2/C3 repeated as Tables XVI and XVII/XVIII are the before (pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a finalization routine respectively. The modified code that is added to the method is highlighted in bold. In the original code sample of

Annexure C1, the finalize method prints "Deleted..." to the computer console on event of finalization (i.e. deletion) of this object. Thus, without management of object finalization in a distributed environment, each machine would re-finalize the same object, thus executing the finalize method more than once for a single globally-named coherent plurality of similar equivalent objects. Clearly this is not what the programmer or user of a single application program code instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the clean-up, deletion, or finalization routine or method. It will be appreciated that the term finalization is typically used in the context of the JAVA language relative to the JAVA virtual machine specification existent at the date of filing of this specification. Therefore, finalization refers to object and/or class cleanup or deletion or reclamation or recycling or any equivalent form of object, class, asset or resource clean-up in the more general sense. The term finalization should therefore be taken in this broader meaning unless otherwise restricted. The changes made (highlighted in **bold**) are the initial instructions that the finalize method executes. These added instructions check if this particular object is the last remaining object of the plurality of similar equivalent objects on the plurality of machines M1, M2...Mn to be marked as finalizable, by calling a routine or procedure to determine the clean-up status of the object to be finalized, such as the "isLastReference()" procedure or method of a DRT 71 performing the steps of 172-176 of FIG. 17 where the determination as to the clean-up status of the particular object is sought, and which determines either a true result or a false result corresponding to whether or not this particular object on this particular machine that is executing the determination procedure is the last of the plurality of machines M1, M2...Mn, each with one of a similar equivalent peer object, to request finalization. Recall that a peer object refers to a similar equivalent object on a different one of the machines, so that for example, in a configuration having eight machines, there will be eight peer objects (i.e. eight similar equivalent objects each on one of eight machines).

The finalization determination procedure or method "isLastReference()" of the DRT 71 can optionally take an argument which represents a unique identifier for this object (See Annexure C3 and Table XVIII). For example, the name of the object that is being considered for finalization, a reference to the object in question being

considered for finalization, or a unique number or identifier representing this object across all machines (or nodes), to be used in the determination of the finalization status of this object or class or other asset. This way, the DRT can support the finalization of multiple objects (or classes or assets) at the same time without becoming confused as to which of the multiple objects are already finalized and which are not, by using the unique identifier of each object to consult the correct record in the finalization table referred to earlier.

The DRT 71 can determine the finalization state of the object in a number of possible ways. Preferably, it (the requesting machine) can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s) if their requested machine's similar equivalent object has been marked for finalization, and if any requested machine replies false indicating that their similar equivalent object is not marked for finalization, then return a false result at return from the "isLastReference()" method indicating that the local similar equivalent object should not be finalized, otherwise return a true result at return from the "isLastReference()" method indicating that the local similar equivalent object can be finalized. Of course different logic schemes for true or false result may alternatively be implemented with the same effect. Alternatively, the DRT 71 on the local machine can consult a shared record table (perhaps on a separate machine (e.g., machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if each of the plurality of similar equivalent objects have been marked for finalization by all requested machines except the current requesting machine.

If the "isLastReference()" method of the DRT 71 returns true then this means that this object has been marked for finalization on all other machines in the virtual or distributed computing environment (i.e. the plurality of machines M1...Mn), and hence, the execution of the finalize method is to proceed as this is considered the last remaining similar equivalent object on the plurality of machines M1, M2...Mn to be marked or declared as finalizable.

On the other hand, if the "isLastReference()" method of the DRT 71 returns false, then this means that the plurality of similar equivalent objects has not been

marked for finalization by all other machines in the distributed environment, as recorded in the shared record table on machine X of the finalization states of objects. In such a case, the finalize method is not to be executed (or alternatively resumed, or continued), as it will potentially invalidate the object on those machine(s) that are continuing to use their similar equivalent object and have yet to mark their similar equivalent object for finalization. Thus, when the DRT returns false, the inserted four instructions at the start of the finalize method prevent execution of the remaining code of the finalize method by aborting the execution of the finalize method through the use of a return instruction, and consequently aborting the Java Virtual Machine's finalization operation for this object.

Given the fundamental concept of testing to determine if a finalization, such as a deletion or clean up, is ready to be carried out on a class, object, or other asset; and if ready carrying out the finalization, and if not ready, then not carrying out the finalization, there are several different ways or embodiments in which this finalization concept, method, and procedure may be implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of a clean up routine modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine, or procedure) with the modified object (or class or asset) inclusive of the now modified clean up routine or routines. Note that there may be one or a plurality of routines corresponding to only one object in the application code or there can be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the cleanup routine(s) that is (are) loaded is binary executable object code. Alternatively, the cleanup routine(s) that is (are) loaded is executable intermediate code.

In one arrangement, which may be termed "master/slave" (or primary/secondary) each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the now modified clean-up routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer

communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads the asset (such as class or object) inclusive of a cleanup routine in unmodified form on machine M2, and then (for example, M2 or each local machine) deletes the unmodified clean up routine that had been present on the machine in whole or part from the asset (such as class or object) and loads by means of a computer communications network the modified code for the asset with the now modified or deleted clean up routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset clean up routine but a deletion of the clean up routine on all machines except one. In one embodiment, the actual code-block of the finalization or cleanup routine is deleted on all machines except one, and this last machine therefore is the only machine that can execute the finalization routine because all other machines have deleted the finalization routine. One benefit of this approach is that no conflict arises between multiple machines executing the same finalization routine because only one machine has the routine.

The process of deleting the clean up routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3...Mn upon

receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code for the asset, class or object between machines M1, M2...Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of finalization or clean up routine(s), but modifies the routine(s) and then loads the asset (such as class or object) consisting of the now modified routine(s). Although one machine, such as the master or primary machine may customize or perform a different modification to the finalization or clean up routine(s) sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of a finalization or clean up routine and all other machines M2, M3, ..., Mn perform a modification to delete the clean up routine of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15 can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3, which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the machines M1, ...,Mn, may send some or all load requests to an additional machine X (See for example the embodiment of FIG. 15), which performs the modification to the application program code 50 (such as

consisting of assets, and/or classes, and/or objects) and inclusive of finalization or clean up routine(s), via any of the afore mentioned methods, and returns the modified application program code inclusive of the now modified finalization or clean-up routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application program code inclusive of the modified routine(s) locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code inclusive of modified finalization or clean-up routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

SYNCHRONIZATION

Turning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Furthermore, the single machine (not a plurality of connected or coupled machines) of FIG. 14, or a more general virtual machine or abstract machine environment such as for example but not limited to an object-oriented virtual machine, is able to readily ensure that multiple different and potentially concurrent uses of specific objects 50X-50Z do not conflict or cause unwanted interactions, when specified by the use of mutual exclusion (e.g. "mutex") operators or operations (inclusive for example of locks, semaphores, monitors, barriers, and the like), such as for example by the programmer's use of a synchronizing or synchronization routine in a computer program written in the JAVA language. As each object exists singularly and only locally (that is locally within the machine within which execution is

occurring) in this example, the single JAVA virtual machine 72 of FIG. 14 executing within this single machine is able to ensure that an object (or several objects) is (are) properly synchronized as defined by the JAVA Virtual Machine and Language Specifications existent at least as of the date of the filing of this patent application, when specified to do so by the application program (or programmer), and thus the object or objects to be synchronized are only utilized by one executing part of potentially multiple executing parts and potentially concurrently executing parts of the executable application code 50 at once or at the same time, such as for example potentially concurrently executing threads or processes. If another executing part and potentially concurrently executing part (such as for example but not limited to a potentially concurrently executing thread or process) of the executable application code 50 wishes to exclusively use the same object whilst that object is the subject of a mutual exclusion operation by a first executing part (e.g. a first thread or process), such as when a second executing part (e.g. a second thread or process) of a multiple part processing machine of FIG. 14 attempts to synchronize on a same object already synchronized by a first executing part, then the possible conflict is resolved by the JAVA virtual machine 72 such that the second and additional executing parts and potentially concurrently executing part or parts of the application program 50 have to wait until the first executing part has finished the execution of its synchronization routine or other mutual exclusion operation. It may be appreciated that in a conventional situation, a second or multiple executing part(s) (i.e. a second or multiple thread(s)) of the application program or program code may want to use the same object in a multiple-thread processing machine of FIG. 14.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International

Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

A similar procedure applies mutatis mutandis (that is, with suitable or necessary alterations) for classes 50A. In particular, the computer programmer (or if and when applicable, an automated or nonautomated computer program generator or generation means) when writing or generating a program using the JAVA language and architecture in a single machine, need only use a synchronization routine or routines in order to provide for this avoidance of conflict or unwanted interaction. Thus a single JAVA virtual machine can keep track of exclusive utilization of the classes and objects (or other asset) and avoid corresponding problems (such as conflict, race condition, unwanted interaction, or other anomalous behaviour due to unexpected critical dependence on the relative timing of events) as necessary in an unobtrusive fashion. The process whereby only one object or class is exclusively used is termed "synchronization" in the JAVA language. In the JAVA language, synchronization may usually be operationalized or implemented in one of three ways or means. The first way or means is through the use of a synchronization method description that is included in the source-code of an application program written in the JAVA language. The second way or means is by the inclusion of a 'synchronization descriptor' in the method descriptor of a compiled application program of the JAVA virtual machine. And the third way or means for performing synchronization are by the use of the instructions monitor enter (e.g., "monitorenter") and monitor exit (e.g., "monitorexit") of the JAVA virtual machine which signify respectively the beginning and ending of a synchronization routine which results in the acquiring or execution of a "lock" (or other mutual exclusion operator or operation), and the releasing or

termination of a "lock" (or other mutual exclusion operator or operation) respectively which prevents an asset being the subject of conflict (or race condition, or unwanted interaction, or other anomalous behaviour due to unexpected critical dependence on the relative timing of events) between multiple and potentially concurrent uses. An asset may for example include a class or an object, as well as any other software /language/ runtime/ platform/ architecture or machine resource. Such resources may include for example, but are not limited to, software programs (such as for example executable software. modules, subprograms, sub-modules, application program interfaces (API), software libraries, dynamically linkable libraries) and data (such as for example data types, data structures, variables, arrays, lists, structures, unions), and memory locations (such as for example named memory locations, memory ranges, address space(s), registers,) and input/output (I/O) ports and/or interfaces, or other machine, computer ,or information appliance resource or asset.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2,…, Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines is provided with a modifier 51 (See in FIG. 5) and realised by or in for example the distributed run time (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2…Mn, or optionally on each one of some subset of the plurality of computers or machines M1, M2…Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or

coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71 ensures that when an executing part (such as a thread or process) of the modified application program 50 running on one or more of the machines exclusively utilizes (e.g., by means of a synchronization routine or similar or equivalent mutual exclusion operator or operation) a particular local asset, such as an objects 50X-50Z or class 50A, no other executing part and potentially concurrently executing part on machines M2...Mn exclusively utilizes the similar equivalent corresponding asset in its local memory at once or at the same time.

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed runtime system 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for

communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

It will therefore be understood in light of the description provided here that the invention further includes any means of implementing thread-safety, regardless of whether it is through the use of locks (lock/unlock), synchronizations, monitors, semphafores, mutexes, or other mechanisms.

It will be appreciated that synchronization means or implies "exclusive use" or "mutual exclusion" of an asset or resource. Conventional structures and methods for implementations of single computers or machines have developed some methods for synchronization on such single computer or machine configurations. However, these conventional structures and methods have not provided solutions for synchronization between and among a plurality of computers, machines, or information appliances.

In particular, whilst one particular machine (say, for example machine M3) is exclusively using an object or class (or any other asset or resource), another machine (say, for example machine M5) may also be instructed by the code it is executing to exclusively use the local similar equivalent object or class corresponding to the similar equivalent object or class on machine M3 at the same time or an overlapping time period. Thus if the same corresponding local similar equivalent objects or classes on each machine M3 and M5 were to be exclusively used by both machines, then the behaviour of the object and application as a whole is undefined – that is, in the absence of proper exclusive use of an object (or class) when explicitly specified by the computer program (programmer), conflict, race conditions, unwanted interactions, anomalous behaviour due to unexpected dependence on the relative timing of events, or permanent inconsistency between the similar equivalent objects on machines M5 and M3 is likely to result. Thus a goal of achieving or providing consistent, coordinated, and coherent operation of synchronization routines (or other mutual exclusion operations) between and amongst a plurality of machines, as required for the simultaneous and coordinated operation of the same application program code on each of the plurality of machines M1, M2...Mn, would not be achieved.

In order to ensure consistent synchronization between and amongst machines M1, M2...Mn the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent a synchronization routine (or other mutual exclusion operation). In the JAVA language, such program steps may for example comprise or consist of an opening monitor enter (e.g. "monitorenter") instruction and one or more closing monitor exit (e.g. "monitorexit") instructions. In one embodiment, a synchronization routine may start with the execution of a "monitorenter" instruction and close with a paired execution of a "monitorexit" instruction.

This analysis or scrutiny of the application code 50 may take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the

application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

Reference is made to the accompanying Annexure D in which: Annexure D1 is a typical code fragment from a synchronization routine prior to modification (e.g., an exemplary unmodified synchronization routine), and Annexure D2 is the same synchronization routine after modification (e.g., an exemplary modified synchronization routine). These code fragments are exemplary only and identify one software code means for performing the modification in an exemplary language. It will be appreciated that other software/firmware or computer program code may be used to accomplish the same or analogous function or operation without departing from the invention.

Annexures D1 and D2 (also reproduced in part in Tables XX and XXI below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a synchronization routine of application program 50 and a post-modification excerpt of the same synchronization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the synchronization method is highlighted in **bold** text. Other embodiments of the invention may provide for code or statements or instructions to be added, amended, removed, moved or reorganized, or otherwise altered.

It is noted that the compiled code in the Annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure D3. The disassembled compiled code that is listed in the Annexure and Table is taken from compiled source code of the file "EXAMPLE.JAVA". In the procedure of Annexure D1 and Table XX, the procedure name "Method void run()" of Step 001 is the name of the displayed disassembled output of the run method of the compiled application code of "example.java". The name "Method void run()" is arbitrary and selected for this example to indicate a typical JAVA method inclusive of a synchronization operation. Overall the method is responsible for incrementing a memory location ("counter") in a thread-safe manner through the use of a synchronization statement and the steps to accomplish this are described in turn.

First (Step 002), the Java Virtual Machine instruction "getstatic #2 <Field java.lang.Object LOCK>" causes the Java Virtual Machine to retrieve the object reference of the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 2nd index of the classfile structure of the application program containing this example run() method and results in a reference to the object (hereafter referred to as LOCK) in the field to be placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the Java Virtual Machine instruction "dup" causes the Java Virtual Machine to duplicate the topmost item of the stack and push the duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the LOCK object at the top of the stack being duplicated and pushed onto the stack.

Next (Step 004), the Java Virtual Machine instruction "astore_1" causes the Java Virtual Machine to remove the topmost item of the stack of the current method frame and store the item into the local variable array at index 1 of the current method frame and results in the topmost LOCK object reference of the stack being stored in the local variable index 1.

Then (Step 005), the Java Virtual Machine instruction "monitorenter" causes the Java Virtual Machine to pop the topmost object off the stack of the current method frame and acquire an exclusive lock on said popped object and results in a lock being acquired on the LOCK object.

The Java Virtual Machine instruction "getstatic #3 <Field int counter>" (Step 006) causes the Java Virtual Machine to retrieve the integer value of the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example run() method and results in the integer value of said field being placed (pushed) on the stack of the current method frame of the currently executing thread.

The Java Virtual Machine instruction "iconst_1" (Step 007) causes the Java Virtual Machine to load an integer value of "1" onto the stack of the current method frame and results in the integer value of 1 loaded onto the top of the stack of the current method frame.

The Java Virtual Machine instruction "iadd" (Step 008) causes the Java Virtual Machine to perform an integer addition of the two topmost integer values of the stack of the current method frame and results in the resulting integer value of the addition operation being placed on the top of the stack of the current method frame.

The Java Virtual Machine instruction "putstatic #3 <Field int counter>" (Step 009) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example run() method and results in the topmost integer value of the stack of the current method frame being stored in the integer field named "counter".

The Java Virtual Machine instruction "aload_1" (Step 010) causes the Java Virtual Machine to load the item in the local variable array at index 1 of the current method frame and store this item on the top of the stack of the current method frame and results in the object reference stored in the local variable array at index 1 being pushed onto the stack.

The Java Virtual Machine instruction "monitorexit" (Step 011) causes the Java Virtual Machine to pop the topmost object off the stack of the current method frame and release the exclusive lock on said popped object and results in the LOCK being released on the LOCK object.

Finally, the Java Virtual Machine instruction "return" (Step 012) causes the Java Virtual Machine to cease executing this run() method by returning control to the

previous method frame and results in termination of execution of this run() method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the synchronization statement enclosing the increment operation of the "counter" memory location ensures that no two or more concurrently execution instances of this run() method will conflict, or otherwise result in unwanted interactions such as a race-condition or other anomalous behaviour due to unexpected critical dependence on the relative timing of the incrementing events performed of the one "counter" memory location. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12 and 13, and concurrently executing two or more instances or occurrences of the run() method each on a different one of the plurality of machines M1, M2...Mn, the mutual exclusion operations of each concurrently executing instance of the run() method would be performed on each corresponding one of the machines without coordination between those machines.

Given the goal of consistent coordinated synchronization operation across a plurality of machines, this prior art arrangement would fail to perform such consistent coordinated synchronization operation across the plurality of machines, as each machine performs synchronization only locally and without any attempt to coordinate their local synchronization operation with any other similar synchronization operation on any one or more other machines. Such an arrangement would therefore be susceptible to conflict or other unwanted interactions (such as race-conditions or other anomalous behaviour due to unexpected critical dependence on the relative timing of the "counter" increment events on each machine) between the machines M1, M2, ..., Mn. Therefore it is the goal of the present invention to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XXI (Annexure D2), the code has been modified so that it solves the problem of consistent coordinated synchronization operation for a plurality of machines M1, M2, ..., Mn, that was not solved in the code example from Table XX (Annexure D1). In this modified run() method code, a "dup" instruction is inserted between the "4 astore_1" and "6 monitorenter" instructions. This causes the Java Virtual Machine to duplicate the topmost item of the stack and

push said duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the LOCK object at the top of the stack being duplicated and pushed onto the stack.

Furthermore, the Java Virtual Machine instruction "invokestatic #23 <Method void acquireLock(java.lang.Object)>" is inserted after the "6 monitorenter" and before the "10 getstatic #3 <Field int counter>" statements so that the Java Virtual Machine pops the topmost item off the stack of the current method frame and invokes the "acquireLock" method, passing the popped item to the new method frame as its first argument. This change is particularly significant because it modifies the run() method to execute the "acquireLock" method and associated operations, corresponding to the "monitorenter" instruction preceding it.

Annexure D1 is a before-modification excerpt of the disassembled compiled form of the synchronization operation of example.java of Annexure D3, consisting of an starting "monitorenter" instruction and ending "monitorexit" instruction. Annexure D2 is an after-modification form of Annexure D1, modified by LockLoader.java of Annexure D6 in accordance with the steps of FIG. 26. The modifications are highlighted in bold.

Table XX. Annexure D1

Step	Annexure D1
001	Method void run()
002	0 getstatic #2 <Field java.lang.Object LOCK>
003	3 dup
004	4 astore_1
005	5 monitorenter
006	6 getstatic #3 <Field int counter>
007	9 iconst_1
008	10 iadd
009	11 putstatic #3 <Field int counter>
010	14 aload_1
011	15 monitorexit
012	16 return

Table XXI. Annexure D2

Step	Annexure D2
001	Method void run()
002	0 getstatic #2 <Field java.lang.Object LOCK>
003	3 dup
004	4 astore_1
004A	**5 dup**
005	6 monitorenter
005A	**7 invokestatic #23 <Method void acquireLock(java.lang.Object)>**
006	10 getstatic #3 <Field int counter>
007	13 iconst_1
008	14 iadd
009	15 putstatic #3 <Field int counter>
010	18 aload_1
010A	**19 dup**
010B	**20 invokestatic #24 <Method void releaseLock(java.lang.Object)>**
011	23 monitorexit
	24 return

The method void acquireLock(java.lang.Object), part of the LockClient code of Annexure D4 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1, ..., Mn to coordinate the execution of the preceding "monitorenter" synchronization operation amongst the machines M1...Mn. The acquireLock method of this example communicates with the LockServer code of Annexure D5 executing on a machine X of FIG. 15, by means of sending an 'acquire lock request' to machine X corresponding to the object being 'locked' (i.e., the object corresponding to the "monitorenter" instruction), which in the context of Table XXI and Annexure D2 is the 'LOCK' object. With reference to FIG. 29, Machine X receives the 'acquire lock request' corresponding to the LOCK object, and consults a table of locks to determine the lock status corresponding to the plurality of similar equivalent objects on each of the machines, which in the case of Annexure D2 is the plurality of similar equivalent LOCK objects.

If all of the plurality of similar equivalent objects on each of the plurality of machines M1...Mn is presently not locked by any other machine M1...Mn, then Machine X will record the object as now locked and inform the requesting machine of the successful acquisition of the lock. Alternatively, if a similar equivalent object is presently locked by another one of the machines M1...Mn, then Machine X will append this requesting machine to a queue of machines waiting to lock this plurality of similar equivalent objects, until such a time as machine X determines this requesting machine can acquire the lock. Corresponding to the successful acquisition of a lock by a requesting machine, a reply is generated and sent to the successful requesting machine informing that machine of the successful acquisition of the lock. Following a receipt of such a message from Machine X confirming the successful acquisition of a requested lock, the acquireLock method and operations terminate execution and return control to the previous method frame, which is the context of Annexure D2 is the executing method frame of the run() method. Until such a time as the requesting machine receives a reply from machine X confirming the successful acquisition of the requested lock, the operation of the acquireLock method and run() method are suspended until such a confirmatory reply is received. Following this return operation, the execution of the run() method then resumes. Exemplary source-code for an embodiment of the acquireLock method is provided in Annexure D4. Annexure D4 also provides additional detail concerning DRT 71 functionality.

Later, the two statements "dup" and "invokestatic #24 <Method void releaseLock(java.lang.Object)>" are inserted into the code stream after the "18 aload_1" statement and before the "23 monitorexit" statement. These two statements cause the Java Virtual Machine to duplicate the item on the stack and then invoke the releaseLock method with the topmost item of the stack as an argument to the method call and result in the modification of the run() method to execute the "releaseLock" method and associated operations, corresponding to the following "monitorexit" instruction, before the procedure exits and returns.

The method void releaseLock(java.lang.Object), part of the LockClient code of Annexure D4 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the following "monitorexit" synchronization operation amongst the machines

M1...Mn. The releaseLock method of this example communications with LockServer code of Annexure D5 executing on a machine X of FIG. 15, by means of sending a "release lock request" to machine X corresponding to the object being "unlocked" (i.e., the object corresponding to the "monitorexit" instruction), which in the context of Table XXI and Annexure D2 is the 'LOCK' object. Corresponding to FIG. 30, machine X receives the "release lock request" corresponding to the LOCK object, and updates the table of locks to indicate the lock status corresponding to the plurality of similar equivalent 'LOCK' objects as now "unlocked". Additionally, if there are other machines awaiting acquisition of this lock, then machine X is able to select one of the awaiting machines to be the new owner of the lock by updating the table of locks to indicate this selected one awaiting machine as the new lock owner, and informing the successful one of the awaiting machines of its successful acquisition of the lock by means of a confirmatory reply. The successful one of the awaiting machines then resumes execution of its synchronization routine. Following the notification to machine X of lock release, the releaseLock method terminates execution and returns control to the previous method frame, which in this instance is the method frame of the run() method. Following this return operation, the execution of the run() method resumes.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of synchronization routines or other mutual exclusion operations between and amongst machines M1...Mn so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as conflicts, unwanted interactions, race-conditions, or anomalous behaviour due to unexpected critical dependence on the relative time of events) does not occur when applying the modified code or procedure.

In the unmodified code sample of Annexure D1, the application program code includes instructions or operations that increment a memory location in local memory (used for a counter) within an enclosing synchronization routine. The purpose of the synchronization routine is to ensure thread-safety of the counter memory increment operation in multi-threaded and multi-processing applications and computer systems. The terms thread-safe or thread-safety refer to code that is either re-entrant or

protected from multiple simultaneous execution by some form of mutual exclusion. Multi-threaded applications in the context of the invention may, for example, include applications operating two or more threads of execution concurrently each on a different machine. Thus, without the management of coordinated synchronization in environments comprising or consisting of a plurality of machines, each running concurrently executing part of a same application program, and with a memory updating and propagation replication means of FIGS. 9, 10, 11, 12, and 13, each computer or computing machine would perform synchronization in isolation, thus potentially incrementing the shared counter at the same time, leading to potential conflicts or unwanted interactions such as race condition(s) and incoherent memory between the machines M1...Mn. It will be appreciated that although this embodiment is described using a shared counter, the use or provision of such shared counter or memory location is optional and not required for the synchronization aspects of the invention. What is advantageous is that the synchronization routine behaves in a manner as the programming language, runtime system, or machine architecture (or any combination thereof) guarantees – that is, stop two parts (for example, two threads) of the application program from executing the same synchronization routine or same mutual exclusion operation or operator concurrently. Clearly consistent, coherent and coordinated synchronization behaviour is what the programmer or user of the application program code 50 expects to happen.

So, taking advantage of the DRT 71, the application code 50 is modified as it is loaded into the machine by changing the synchronization routine. It will be appreciated in light of the description provided here that the modifications made on each machine may generally be similar in-so-far as they should advantageously achieve a consistent end result of coordinated synchronization operation amongst all the machines; however, given the broad applicability of the inventive synchronization method and associated procedures, the nature of the modifications may generally vary without altering the effect produced. For example, in a simple variation, one or more additional instructions or statements may be inserted, such as for example a "no-operation" (nop) type instruction into the application will mean the modifications made are technically different, but the modified code still conforms to the invention. Embodiments of the invention may for example, implement the changes by means of

program transformation, translation, various forms of compilation, instrumentation, or by other means described herein or known in the art. The changes made (highlighted in bold text) are the starting or initial instructions and the ending instructions that the synchronization routine executes, and which correspond to the entry (start) and exit (finish) of the synchronization routine respectively. These added instructions (or modified instruction stream) act to coordinate the execution of the synchronization routine amongst the multiple concurrently executing instances or occurrences of the modified run method executing on each one of, or some subset of, the plurality of machines M1...Mn, by invoking the acquireLock method corresponding to the start of execution of the synchronization routine, and by invoking the releaseLock method corresponding to the finish of execution of the synchronization routine, thereby providing consistent coordinated operation of the synchronization routine (or other mutual exclusion operation or operator) as required for the simultaneous operation of the modified application program code that is running on or across the plurality of machines M1, M2,..., Mn. This also advantageously provides for operation of the one application program in a coordinated manner across the machines.

The acquire lock (e.g. "acquireLock()") method of the DRT 71 takes an argument "(java.lang.Object)" which represents a reference to (or some other unique identifier for) the particular local object for which the global lock is desired (See Annexure D2 and Table XXI), and is to be used in acquiring a global lock across the plurality of similar equivalent objects on the other machines corresponding to the specified local object. The unique identifier may, for example be the name of the object, a reference to the object in question, or a unique number representing the plurality of similar equivalent objects across all nodes. By using a globally unique identifier across all connected machines to represent the plurality of similar equivalent objects on the plurality of machines, the DRT can support the synchronization of multiple objects at the same time without becoming confused as to which of the multiple objects are already synchronized and which are not as might be the case if object (or class) identifiers were not unique, by using the unique identifier of each object to consult the correct record in the shared synchronization table.

A further advantage of using a global identifier here is as a form of 'meta-name' for all the similar equivalent local objects on each one of the machines.

For example, rather than having to keep track of each unique local name of each similar equivalent local object on each machine, one may instead define a global name (e.g., "globalname7787") which each local machine in turn maps to a local object (e.g., "globalname7787" points to object "localobject456" on machine M1, and "globalname7787" points to object "localobject885" on machine M2, and "globalname7787" points to object "localobject111" on machine M3, and so forth). It thereafter is easier to simply say "acquire lock for globalname7787" which is then translated on machine 1 (M1) to mean "acquire lock for localobject456", and is translated on machine 2 (M2) to mean "acquire lock for localobject885", and so on.

The shared synchronization table that may optionally be used is a table, other storage means, or any other data structure that stores an object (and/or class or other asset) identifier and the synchronization status (or locked or unlocked status) of each object (and/or class or other asset). The table or other storage means operates to relate an object (and/or class or other asset, or a plurality of similar equivalent objects or classes or assets) to a status of either locked or unlocked or some other physical or logical indication of a locked state and an unlocked state. For example: the table (or any other data structure one cares to employ) may advantageously include a named object identifier and a record indicating if a named object (i.e., "globalname7787") is locked or unlocked. In one embodiment, the table or other storage means stores a flag or memory bit, wherein when the flag or memory bit stores a "0" the object is unlocked and when the flag or memory bit stores a "1" the object is locked. Clearly, multiple bit or byte storage may be used and different logic sense or indicators may be used without departing from the invention.

The DRT 71 can determine the synchronization state of the object in any one of a number of ways. Recall, for example that the invention may include any means of implementing thread-safety, regardless of whether it is through the use of locks (lock/unlock), synchronizations, monitors, semphafores, mutexes, or other mechanisms. These means stop or limit concurrently executing parts of a single application program in order to guarantee consistency according to the rules of synchronization, locks, or the like. Preferably, it can ask each machine in turn if their local similar equivalent object (or class or other asset or resource) corresponding to the object being sought to be locked is presently synchronized, and if any machine

replies true, then to pause execution of the synchronization routine and wait until that presently synchronized similar equivalent object on the other machine is unsynchronised, otherwise synchronize this object locally and resume execution of the synchronization routine. Each machine may implement synchronization (or mutual exclusion operations or operators) in its own way and this may be different in the different machines. Therefore, although some exemplary implementation details are provided, ultimately how synchronization (or mutual exclusion operations) is (are) implemented, or precisely how synchronization or mutual exclusion status (or locked/unlocked status) is recorded in memory or other storage means, is not critical to the invention. By unsynchronized we generally mean unlocked or otherwise not subject to a mutual exclusion operation, and by synchronized we generally mean locked and subject to a mutual exclusion operation.

Alternatively, the DRT 71 on each local machine can consult a shared record table (perhaps on a separate machine (for example, on machine X which is different from machines M1, M2, ..., Mn)), or can consult a coherent shared record table on each one of the local machines, or a shared database established in a memory or other storage, to determine if this object has been marked or identified as synchronized (or "locked") by any machine and if so, then wait until the status of the object is changed to "unlocked" and then acquire the lock on this machine, otherwise acquire the lock by marking the object as locked (optionally by this machine) in the shared lock table.

In the situation where the shared record table is consulted, this may be considered as a variation of a shared database or data structure, where each machine has a local copy of a shared table (that is a replica of a shared table) with is updated to maintain coherency across the plurality of machines M1, ..., Mn.

In one embodiment, the shared record table refers to a shared table accessible by all machines M1, ..., Mn, that may for example be defined or stored in a commonly accessibly database such that any machine M1, ..., Mn can consult or read this shared database table for the locked or unlocked status of an object. A further alternative arrangement is to implement a shared record table as a table in the memory of an additional machine (which we call "machine X") which stores each object identification name and its lock status, and serves as the central repository which all

other machines M1, ..., Mn consult to determine locked status of similar equivalent objects.

In any of these different alternative implementations, the manner in which a one of, or a plurality of, similar equivalent objects is marked or identified as being synchronized (or locked) or unsynchronized (or unlocked) is relatively unimportant, and various stored memory bits or bytes or flags may be utilized as are known in the art to identify either one of the two possible logic states. It will also be appreciated that in the present embodiment, that synchronized is largely synonymous with locked and unsynchronized is largely synonymous with unlocked. These same considerations apply for classes as well as for other assets or resources.

Recall that the DRT 71 is responsible for determining the locked status for an object (or class, or other asset, corresponding to a plurality of similar equivalent objects or classes or assets) seeking to be locked before allowing the synchronization routine corresponding to the acquisition of that lock to proceed. In the exemplary embodiment described here, the DRT consults the shared synchronization record table which in one embodiment resides on an special "machine X", and therefore the DRT needs to communicate via the network or other communications link or path with this machine X to enquire as to and determine the locked (or unlocked) status of the object (or class or other asset corresponding to a plurality of similar equivalent objects or classes or assets).

If the DRT on the local machine that is trying to execute a synchronization routine or other mutual exclusion operation determines that no other machine currently has a lock for this object (i.e., no other machine has synchronized this object) or any other one of a plurality of similar equivalent objects, then to acquire the lock for this object corresponding to the plurality of similar equivalent objects on all other machines, for example by means of modifying the corresponding entry in a shared table of locked states for the object sought to be locked or alternatively, sequentially acquiring the lock on all other similar equivalent objects on all other machines in addition to the current machine. Note that the intent of this procedure is to lock the plurality of similar equivalent objects (or classes or assets) on all the other machines M1, ..., Mn so that simultaneous or concurrent use of any similar equivalent objects by two or more machines is prevented, and any available approach

may be utilized to accomplish this coordinated locking. For example, it does not matter if machine M1 instructs M2 to lock its similar equivalent local object, then instructs M3 to lock its similar equivalent local object, and then instructs M4 and so on; or if M1 instructs M2 to lock its similar equivalent local object, and then M2 instructs M3 to lock its similar equivalent local object, and then M3 instructs M4 to lock its similar equivalent local object, and so forth, what is being sought is the locking of the similar equivalent objects on all other machines so that simultaneous or concurrent use any similar equivalent objects by two or more machines is prevented. Only once this machine has successfully confirmed that no other machine has currently locked a similar equivalent object, and this machine has correspondingly locked its locally similar equivalent object, can the execution of the synchronization routine or code-block begin.

On the other hand, if the DRT 71 within the machine about to execute a synchronization routine (such as machine M1) determines that another machine, such as machine M4 has already synchronized a similar equivalent object, then this machine M1 is to postpone continued execution of the synchronization routine (or code-block) until such a time as the DRT on machine M1 can confirm than no other machine (such as one of machines M2, M3, M4, or M5, ..., Mn) is presently executing a synchronize routine on a corresponding similar equivalent local object, and that this machine M1 has correspondingly synchronized its similar equivalent object locally. Recall that local synchronization refers to prior art conventional synchronization on a single machine, whereas global or coordinated synchronization refers to coordinated synchronization of, across and/or between similar equivalent local objects each on a one of the plurality of machines M1...Mn. In such a case, the synchronization routine (or code-block) is not to continue execution until this machine M1 can guarantee that no other machine M2, M3, M4, ..., Mn is executing a synchronization routine corresponding to the local similar equivalent object being sought to be locked, as it will potentially corrupt the object across the participating machines M1, M2, M3, ..., Mn due to susceptibility to conflicts or other unwanted interactions such as race-conditions, and the like problems resulting from the concurrent execution of synchronization routines. Thus, when the DRT determines that this object, or a similar equivalent object on another machine, is presently

"locked", say by machine M4 (relative to all other machines), the DRT on machine M1 pauses execution of the synchronization routine by pausing the execution of the acquire lock (e.g., "acquireLock()") operation until such a time as a corresponding release lock (e.g., "releaseLock()") operation is executed by the present owner of the lock (e.g., machine M4).

Thus, on execution of a release lock (e.g. "releaseLock()") operation, the machine M4 which presently "owns" or holds a lock (i.e., is executing a synchronization routine) indicates the close of its synchronization routine, for example by marking this object as "unlocked" in the shared table of locked states, or alternatively, sequentially releasing locks acquired on all other machines. At this point, a different machine waiting to begin execution of a paused synchronization statement can then claim ownership of this now released lock by resuming execution of its postponed (i.e. delayed) "acquireLock()" operation, for example, by marking itself as executing a lock for this similar equivalent object in the shared table of synchronization states, or alternatively, sequentially acquiring local locks of similar equivalent objects on each of the other machines. It is to be understood that the resumed execution of the acquire lock (e.g., "acquireLock") operation is to be inclusive of the optional resumption of execution of the acquire lock (e.g., "acquireLock") method at the point that execution was paused, as well as the alternative optional arrangement wherein the execution of the acquire lock (e.g., "acquireLock") operation is repeated so as to re-request the lock. Again, these same considerations also apply for classes and more generally to any asset or resource.

So, according to at least one embodiment and taking advantage of the operation of the DRT 71, the application code 50 is modified as it is loaded into the machine by changing the synchronization routine (consisting of at least a beginning "acquire lock" type instruction (such as a JAVA "monitorenter" instruction) and an ending "release lock" type instruction (such as a JAVA "monitorexit" instruction). "Acquire lock" type instructions commence operation or execution of a mutual exclusion operation, generally corresponding to a particular asset such as a particular memory location or machine resource, and result in the asset corresponding to the mutual exclusion operation being locked with respect to some or all modes of simultaneous or concurrent use, execution or operation. "Release lock" type

instructions terminate or otherwise discontinue operation or execution of a mutual exclusion operation, generally corresponding to a particular asset such as a particular memory location or machine resource, and result in the asset corresponding to the mutual exclusion operation being unlocked with respect to some or all modes of simultaneous or concurrent use, execution or operation. The changes made (highlighted in bold) are the modified instructions that the synchronization routine executes. These added instructions for example check if this lock has already been acquired by another machine. If this lock has not been acquired by another machine, then the DRT of this machine notifies all other machines that this machine has acquired the specified lock, and thereby stopping the other machines from executing synchronization routines corresponding to this lock.

The DRT 71 can determine and record the lock status of similar equivalent objects, or other corresponding memory location or machine or software resource on a plurality of machines, in many ways, such as for example, by way of illustration but not limitation:

1. Corresponding to the entry to a synchronization routine by Machine M1, the DRT of machine M1 individually consults or communicates with each machine to ascertain if this global lock is already acquired by any other Machine M2, ..., Mn different from itself. If this global lock corresponding to this asset or object is or has already been acquired by another one of the machines M2, ..., Mn then the DRT of Machine M1 pauses execution of the synchronization routine on machine M1 until all other machines no longer own a global lock on this asset or object (that is to say that none of the other machines any longer own a global lock corresponding to this asset or object), at which point machine M1 can successfully acquire the global lock such that all other machines M2, ..., Mn must now wait for machine M1 to release the global lock before a different machine can in turn acquire it. Otherwise, when it is determined that this global lock corresponding to this asset or object has not already been acquired by another machine M2, ..., Mn the DRT continues execution of the synchronization routine, and such that all other machines M2, ..., Mn must now wait for machine M1 to release the global lock before a different machine can in turn acquire it.

Alternatively, 2. Corresponding to the entry to a synchronization routine, the DRT consults a shared table of records (for example a shared database, or a copy of a shared table on each of the participating machines) which indicate if any machine currently "owns" this global lock. If so, the DRT then pauses execution of the synchronization routine on this machine until no machine owns a global lock on a similar equivalent object. Otherwise the DRT records this machine in the shared table (or tables, if there are multiple tables of records, e.g., on multiple machines) as the owner of this global lock, and then continues executing the synchronization routine.

Similarly, when a global lock is released, that is to say, when the execution of a synchronization routine is to end, the DRT can "un-record", alter the status indicator, and/or reset the global lock status of machines in many alternative ways, for example by way of illustration but not limitation:

1. Corresponding to the exit to a synchronization routine, the DRT individually notifies each other machine that it no longer owns the global lock.

Alternatively,

2. Corresponding to the exit to a synchronization routine, the DRT updates the record for this globally locked asset or object (such as for example a plurality of similar equivalent objects or assets) in the shared table(s) of records such that this machine is no longer recorded as owning this global lock.

Still further, the DRT can provide an acquire global lock queue to queue machines needing to acquire a global lock in multiple alternative ways, for example by way of illustration but not limitation:

1. Corresponding to the entry to a synchronization routine by Machine M1 say, the DRT of machine M1 notifies the present owning machine (say Machine M4) of the global lock that machine M1 would like to or needs to acquire the corresponding global lock upon release by the current owning machine in order to perform an operation. The specified machine M4, if there are no other waiting machines, then stores a record of the requesting machine's (i.e., machine M1) interest or request in a table or list, such that machine M4 may know subsequent to releasing the corresponding global lock that the machine M1 recorded in the table or list is waiting to acquire the same global lock, which, following the exit of the synchronization routine corresponding to the global lock held by machine M4, then

notifies the waiting machine (i.e. machine M1) specified in the record of waiting machines, that the global lock can be acquired, and thus machine M1 can proceed to acquire the global lock and continue executing its own synchronization routine.

2. Corresponding to the entry to a synchronization routine by machine M1 say, the DRT notifies the present owner of the global lock, say machine M4, that a specific machine (say machine M1) would like to acquire the lock upon release by that machine (i.e., machine M4). That machine M4, if it finds after consulting its records of waiting machines for this locked object, finds that there are already one or more other machines (say machines M2 and M7) waiting, then either appends machine M1 to the end of the list of machines M2 and M7 wanting to acquire this locked object, or alternatively, forwards the request from M1 to the first waiting machine (i.e., machine M2), or any other machine waiting (i.e., machine M7), which then, in turn, records machine M1 in their table or records of waiting machines.

In the example above, for example, the records may be kept on Machine M4 and store a queue or other ordered or indexed list of machines waiting to acquire the lock after Machine M4 releases the lock it holds. This list or queue may then be used or referenced by M4 so that M4 can pass the lock on to other machines in accordance with the order of request or any other prioritization scheme. Alternatively, the list may be unordered, and machine M4 may pass the global lock on to any machine in the list or record.

3. Corresponding to the entry to a synchronization routine, the DRT records itself in a shared table(s) of records (for example, a table stored in a shared database accessible by all machines, or multiple separate tables which are substantially similar).

Still further or in the alternative, the DRT 71 can notify other machines queued to acquire this global lock corresponding to the exit of a synchronization routine by this machine in the following alternative ways, for example:

1. Corresponding to the exit of a synchronization routine, the DRT notifies one of the awaiting machines (for example, this first machine in the queue of waiting machines) that the global lock is released,

2. Corresponding to the exit of a synchronization routine, the DRT notifies one of the awaiting machines (for example, the first machine in the queue of waiting

machines) that the global lock is released, and additionally, provides a copy of the entire queue of machines (for example, the second machine and subsequent machines awaiting for this global lock). This way, the second machine inherits the list of waiting machines from the first machine, and thereby ensures the continuity of the queue of waiting machines as each machine in turn down the list acquires and subsequently releases the same global lock.

During the abovementioned scrutiny, "monitorenter" and "monitorexit" instructions (or methods) are initially looked for and, when found, a modifying code is inserted so as to give rise to a modified synchronization routine. This modified routine additionally acquires and releases the global lock. There are several different modes whereby this modification and loading can be carried out.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2...Mn are as before and run the same application code 50 (or codes) on all machines M1....Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the synchronization of structures, assets, and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably but optionally operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2...Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After loading 161 has been commenced, the instructions to be executed are considered in sequence and all synchronization routines are detected as indicated in step 162. In the JAVA language these are the "monitorenter" and "monitorexit" instructions, and methods

marked as synchronized in the method descriptor. Other languages use different terms.

Where a synchronization routine is detected 162, it is modified in step 163 in order to perform consistent, coordinated, and coherent synchronization operation (or other mutual exclusion operation) across the plurality of machines M1...Mn, typically by inserting further instructions into the synchronization (or other mutual exclusion) routine to, for example, coordinate the operation of the synchronization routine amongst and between similar equivalent synchronization or other mutual exclusion operations on other one or more of the plurality of machines M1...Mn, so that no two or more machines execute a similar equivalent synchronization or other mutual exclusion operation at once or overlapping. Alternatively, the modifying instructions may be inserted prior to the routine, such as for example prior to the instruction(s) or operation(s) related to a synchronization routine. Once the modification step 163 has been completed the loading procedure continues by loading the modified application code in place of the unmodified application code, as indicated in step 164. The modifications preferably take the form of an "acquire lock on all other machines" operation and a "release lock on all other machines" modification as indicated at step 163.

FIG. 27 illustrates a particular form of modification. Firstly, the structures, assets or resources (in JAVA termed classes or objects eg 50A, 50X-50Y) or more generally "locks" to be synchronized have already been allocated a name or tag (for example a global name or tag) which can be used to identify corresponding similar equivalent local objects, or assets, or resources, or locks on each of the machines M1....Mn, as indicated by step 172. This preferably happens when the classes or objects are originally initialized. This is most conveniently done via a table maintained by server machine X. This table also includes the synchronization status of the class or object or lock. It will be understood that this table or other data structure may store only the synchronization status, or it may store other status or information as well. In the preferred embodiment, this table also includes a queue arrangement which stores the identities of machines which have requested use of this asset or lock.

As indicated in step 173 of FIG. 27, next an "acquire lock" request is sent to machine X, after which, the sending machine awaits for confirmation of lock acquisition as shown in step 174. Thus, if the global name is already locked (i.e. a corresponding similar local asset is in exclusive use by another machine other than the machine proposing to acquire the lock) then this means that the proposed synchronization routine of the corresponding object or class or asset or lock should be paused until the corresponding object or class or asset or lock is unlocked by the current owner.

Alternatively, if the global name is not locked, this means that no other machine is exclusively using a similar equivalent class, object, asset or lock, and confirmation of lock acquisition is received straight away. After receipt of confirmation of lock acquisition, execution of the synchronization routine is allowed to continue, as shown in step 175.

FIG. 28 shows the procedures followed by the application program executing machine which wishes to relinquish a lock. The initial step is indicated at step 181. The operation of this proposing machine is temporarily interrupted by steps 183, 184 until the reply is received from machine X, corresponding to step 184, and execution then resumes as indicated in step 185. Optionally, and as indicated in step 182, the machine requesting release of a lock is made to lookup the "global name" for this lock preceding a request being made to machine X. This way, multiple locks on multiple machines may be acquired and released without interfering with one another.

FIG. 29 shows the activity carried out by machine X in response to an "acquire lock" enquiry (of FIG. 27). After receiving an "acquire lock" request at step 191, the lock status is determined at steps 192 and 193 and, if no - the named resource is not free or otherwise "locked", the identity of the enquiring machine is added at step 194 to (or forms) the queue of awaiting acquisition requests. Alternatively, if the answer is yes - the named resource is free and "unlocked" - the corresponding reply is sent at step 197. The waiting enquiring machine is then able to execute the synchronization routine accordingly by carrying out step 175 of FIG. 27. In addition to the yes response, the shared table is updated at step 196 so that the status of the globally named asset is changed to "locked".

FIG. 30 shows the activity carried out by machine X in response to a "release lock" request of FIG. 28. After receiving a "release lock" request at step 201, machine X optionally, and preferably, confirms that the machine requesting to release the global lock is indeed the current owner of the lock, as indicated in step 202. Next, the queue status is determined at step 203 and, if no-one is waiting to acquire this lock, machine X marks this lock as "unowned" (or "unlocked") in the shared table, as shown in step 207, and optionally sends a confirmation of release back to the requesting machine, as indicated by step 208. This enables the requesting machine to execute step 185 of FIG. 28.

Alternatively, if yes – that is, other machines are waiting to acquire this lock - machine X marks this lock as now acquired by the next machine in the queue, as shown in step 204, and then sends a confirmation of lock acquisition to the queued machine at step 205, and consequently removes the new lock owner from the queue of waiting machines, as indicated in step 206.

Given the fundamental concept of modifying the synchronization routines (or other mutual exclusion operations or operators) to coordinate operation between and amongst a plurality of machines M1....Mn, there are several different ways or embodiments in which this coordinated, coherent and consistent synchronization (or other mutual exclusion) operation concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (for example a class or object) inclusive of a synchronization routine(s), modifies it, and then loads each of the other machines M1, M3...Mn (either sequentially, or simultaneously or according to any other order, routine, or procedure) with the modified asset (or class or object) inclusive of the new modified synchronization routine(s). Note that there may be one or a plurality of routine(s) corresponding to only one object in the application code, or there may be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the synchronization routine(s) that is (are) loaded is binary executable object code. Alternatively, the synchronization routine(s) that is (are) loaded is executable intermediate code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified synchronization routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads asset (such as class or object) inclusive of an (or even one or more) synchronization routine in unmodified form on machine M2, and then (for example, machine M2 or each local machine) modifies the class (or object or asset) by deleting the synchronization routine in whole or part from the asset (or class or object) and loads by means of a computer communications network or other communications link or path the modified code for the asset with the now modified or deleted synchronization routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset synchronization routine but a deletion of the synchronization routine on all machines except one.

The process of deleting the synchronization routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3, ..., Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code (including for example, the modified code) for the asset, class or object between machines M1, M2,..., Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more synchronization routines, but modifies the routines and then loads the asset (such as class or object) consisting of the now modified routines. Although one machine, such as the master or primary machine may customize or perform a different modification to the synchronization routine sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of one or more synchronization routines and all other machines M2, M3, ..., Mn perform a modification to delete the synchronization routine(s) of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies

M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the machines M1 to Mn, may send some or all load requests to an additional machine X (see for example the embodiment of FIG. 15), which performs the modification to the application code 50 inclusive of an (and possibly a plurality of) synchronization routine(s) via any of the afore mentioned methods, and returns the modified application code inclusive of the now modified synchronization routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application code inclusive of the modified routines locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code 50 inclusive of modified synchronization routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

EMBODIMENT INCLUDING MEMORY MANAGEMENT AND REPLICATION OBJECT INITIALIZATION, FINALIZATION, AND SYNCHRONIZATION

Having now described structures, procedures, computer program code and tools, and other aspects and features of a multiple computer system and computing method utilizing at least one of memory management and replication object initialization, finalization, and synchronization it may readily be appreciated that these may also optionally but advantageously be applied in any combination

It may also be appreciated that the memory management, initialization, finalization, and/or synchronization aspects of the invention may be implemented or applied serially or sequentially or in parallel. For example, where the code is being scrutinized or analysed to identify or detect particular code sections relevant to initialization, that same analysis or scrutinization may also attempt to identify or detect code sections relevant to finalization (or synchronization for example). Alternatively, separate sequential (or possibly overlapping) analysis and scrutiny may be utilized to separately detect code relevant to initialization and finalization and synchronization. Any required modification to the code may also be performed in combination or separately, and furthermore, portions may be performed together while other portions are performed separately.

Having now described aspects of the memory management and replication, initialization, finalization, and synchronization, attention is now directed to an exemplary operational scenario illustrating the manner in which application programs on two computers may simultaneously execute the same application program in a consistent, coherent manner.

In this regard, attention is directed to FIGS. 31-33, two laptop computers 101 and 102 are illustrated. The computers 101 and 102 are not necessarily identical and indeed, one can be an IBM or IBM-clone and the other can be an APPLE computer. The computers 101 and 102 have two screens 105, 115 two keyboards 106, 116 but a single mouse 107. The two machines 101, 102 are interconnected by a means of a single coaxial cable or twisted pair cable 314.

Two simple application programs are downloaded onto each of the machines 101, 102, the programs being modified as they are being loaded as described above. In this embodiment the first application is a simple calculator program and results in the image of a calculator 108 being displayed on the screen 105. The second program is a graphics program which displays four coloured blocks 109 which are of different colours and which move about at random within a rectangular box 310. Again, after loading, the box 310 is displayed on the screen 105. Each application operates independently so that the blocks 109 are in random motion on the screen 105 whilst numerals within the calculator 108 can be selected (with the mouse 107) together with a mathematical operator (such as addition or multiplication) so that the calculator 108 displays the result.

The mouse 107 can be used to "grab" the box 310 and move same to the right across the screen 105 and onto the screen 115 so as to arrive at the situation illustrated in FIG. 32. In this arrangement, the calculator application is being conducted on machine 101 whilst the graphics application resulting in display of box 310 is being conducted on machine 102.

However, as illustrated in FIG. 33, it is possible by means of the mouse 107 to drag the calculator 108 to the right as seen in FIG. 32 so as to have a part of the calculator 108 displayed by each of the screens 105, 115. Similarly, the box 310 can be dragged by means of the mouse 107 to the left as seen in FIG. 32 so that the box 310 is partially displayed by each of the screens 105, 115 as indicated FIG. 33. In this configuration, part of the calculator operation is being performed on machine 101 and part on machine 102 whilst part of the graphics application is being carried out the machine 101 and the remainder is carried out on machine 102.

FURTHER DESCRIPTION

The foregoing describes only some embodiments of the present invention and modifications, obvious to those skilled in the art, can be made thereto without departing from the scope of the present invention. For example, reference to JAVA includes both the JAVA language and also JAVA platform and architecture.

In all described instances of modification, where the application code 50 is modified before, or during loading, or even after loading but before execution of the

unmodified application code has commenced, it is to be understood that the modified application code is loaded in place of, and executed in place of, the unmodified application code subsequently to the modifications being performed.

Alternatively, in the instances where modification takes place after loading and after execution of the unmodified application code has commenced, it is to be understood that the unmodified application code may either be replaced with the modified application code in whole, corresponding to the modifications being performed, or alternatively, the unmodified application code may be replaced in part or incrementally as the modifications are performed incrementally on the executing unmodified application code. Regardless of which such modification routes are used, the modifications subsequent to being performed execute in place of the unmodified application code.

An advantage of using a global identifier in the invention described is as a form of 'meta-name' or 'meta-identity' for all the similar equivalent local objects (or classes, or assets or resources or the like) on each one of the plurality of machines M1, ..., Mn. For example, rather than having to keep track of each unique local name or identity of each similar equivalent local object on each machine of the plurality of similar equivalent objects, one may instead define or use a global name corresponding to the plurality of similar equivalent objects on each machine (e.g., "globalname7787"), and with the understanding that each machine relates the global name to a specific local name or object (e.g., "globalname7787" corresponds to object "localobject456" on machine M1, and "globalname7787" corresponds to object "localobject885" on machine M2, and "globalname7787" corresponds to object "localobject111" on machine M3, and so forth).

Those skilled in the programming arts will be aware that when additional code or instructions is/are inserted into an existing code or instruction set to modify same, the existing code or instruction set may well require further modification (such as for example, by re-numbering of sequential instructions) so that offsets, branching, attributes, mark up and the like are catered for.

Similarly, in the JAVA language memory locations include, for example, both fields and array types. The above description deals with fields and the changes required for array types are essentially the same mutatis mutandis. Also the present

invention is equally applicable to similar programming languages (including procedural, declarative and object orientated) to JAVA including Micrsoft.NET platform and architecture (Visual Basic, Visual C/C++, and C#) FORTRAN, C/C++, COBOL, BASIC etc.

The abovementioned arrangement, in which the JAVA code which updates memory locations or field values is modified, is based on the assumption that either the runtime system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and Java) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will ordinarily update memory on the local machine (say M2) but not on any corresponding other machines (M1, M3 ...Mn). It is possible to leave the JAVA code which updates memory locations or field values unamended and instead amend the LINUX or HOTSPOT routine which updates memory locally, so that it correspondingly updates memory on all other machines as well. In order to embrace such an arrangement the term "updating propagation routine" used herein in conjunction with maintaining the memory of all machines M1...Mn essentially the same, is to be understood to include within its scope both the JAVA putfield and putstatic instructions and related operations and the "combination" of the JAVA putfield and putstatic operations and the LINUX or HOTSPOT code fragments which perform memory updating.

The abovementioned embodiment in which the code of the JAVA initialisation routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will call the JAVA initialisation routine. It is possible to leave the JAVA initialisation routine unamended and instead amend the LINUX or HOTSPOT routine which calls the JAVA initialisation routine, so that if the object or class is already loaded, then the JAVA initialisation routine is not called. In order to embrace such an arrangement the term "initialisation routine" is to be understood to include within its scope both the JAVA initialisation routine and the "combination" of the JAVA initialisation routine and the LINUX or HOTSPOT code fragments which call or initiates the JAVA initialisation routine.

The abovementioned embodiment in which the code of the JAVA finalisation or clean up routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will call the JAVA finalisation routine. It is possible to leave the JAVA finalisation routine unamended and instead amend the LINUX or HOTSPOT routine which calls the JAVA finalisation routine, so that if the object or class is not to be deleted, then the JAVA finalisation routine is not called. In order to embrace such an arrangement the term "finalisation routine" is to be understood to include within its scope both the JAVA finalisation routine and the "combination" of the JAVA finalisation routine and the LINUX or HOTSPOT code fragments which call or initiate the JAVA finalisation routine.

The abovementioned embodiment in which the code of the JAVA synchronization routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will normally acquire the lock on the local machine (say M2) but not on any other machines (M1, M3 ...Mn). It is possible to leave the JAVA synchronization routine unamended and instead amend the LINUX or HOTSPOT routine which acquires the lock locally, so that it correspondingly acquires the lock on all other machines as well. In order to embrace such an arrangement the term "synchronization routine" is to be understood to include within its scope both the JAVA synchronization routine and the "combination" of the JAVA synchronization routine and the LINUX or HOTSPOT code fragments which perform lock acquisition and release.

The terms object and class used herein are derived from the JAVA environment and are intended to embrace similar terms derived from different environments such as dynamically linked libraries (DLL), or object code packages, or function unit or memory locations.

Various means are described relative to embodiments of the invention, including for example but not limited to lock means, distributed run time means, modifier or modifying means, propagation means, distribution update means, counter

means, synchronization means, and the like. In at least one embodiment of the invention, any one or each of these various means may be implemented by computer program code statements or instructions (possibly including by a plurality of computer program code statements or instructions) that execute within computer logic circuits, processors, ASICs, microprocessors, microcontrollers, or other logic to modify the operation of such logic or circuits to accomplish the recited operation or function. In another embodiment, any one or each of these various means may be implemented in firmware and in other embodiments such may be implemented in hardware. Furthermore, in at least one embodiment of the invention, any one or each of these various means may be implemented by an combination of computer program software, firmware, and/or hardware.

Any and each of the aforedescribed methods, procedures, and/or routines may advantageously be implemented as a computer program and/or computer program product stored on any tangible media or existing in electronic, signal, or digital form. Such computer program or computer program products comprising instructions separately and/or organized as modules, programs, subroutines, or in any other way for execution in processing logic such as in a processor or microprocessor of a computer, computing machine, or information appliance; the computer program or computer program products modifying the operation of the computer on which it executes or on a computer coupled with, connected to, or otherwise in signal communications with the computer on which the computer program or computer program product is present or executing. Such computer program or computer program product modifying the operation and architectural structure of the computer, computing machine, and/or information appliance to alter the technical operation of the computer and realize the technical effects described herein.

The invention may therefore includes a computer program product comprising a set of program instructions stored in a storage medium or exiting electronically in any form and operable to permit a plurality of computers to carry out any of the methods, procedures, routines, or the like as described herein including in any of the claims.

Furthermore, the invention may include a plurality of computers interconnected via a communication network or other communications ink or path and each operable to substantially simultaneously or concurrently execute the same or a different portion of an application program code written to operate on only a single computer on a corresponding

different one of computers, wherein the computers being programmed to carry out any of the methods, procedures, or routines described in the specification or set forth in any of the claims, or being loaded with a computer program product.

The term "comprising" (and its grammatical variations) as used herein is used in the inclusive sense of "having" or "including" and not in the exclusive sense of "consisting only of".

MODIFIED COMPUTER ARCHITECTURE WITH
COORDINATED OBJECTS

ANNEXURES

A, B, C, D

Annexure A

The following are program listings in the JAVA language:

A1. This first excerpt is part of an illustration of the modification code of the modifier 51 in accordance with steps 92 and 103 of FIG. 10. It searches through the code array of the application program code 50, and when it detects a memory manipulation instruction (i.e. a putstatic instruction (opcode 178) in the JAVA language and virtual machine environment) it modifies the application program code by the insertion of an "alert" routine.

```
// START
byte[] code = Code_attribute.code;      // Bytecode of a given method in a
                                        // given classfile.

int code_length = Code_attribute.code_length;

int DRT = 99;       // Location of the CONSTANT_Methodref_info for the
                    // DRT.alert() method.

for (int i=0; i<code_length; i++){

   if ((code[i] & 0xff) == 179){ // Putstatic instruction.

      System.arraycopy(code, i+3, code, i+6, code_length-(i+3));

      code[i+3] = (byte) 184;     // Invokestatic instruction for the
                                  // DRT.alert() method.

      code[i+4] = (byte) ((DRT >>> 8) & 0xff);

      code[i+5] = (byte) (DRT & 0xff);

   }

}
// END
```

A2. This second excerpt is part of the DRT.alert() method and implements the step of 125 and arrow of 127 of FIG. 12. This DRT.alert() method requests one or more threads of the DRT processing environment of Fig. 12 to update and propagate the value and identity of the changed memory location corresponding to the operation of Annexure A1.

```
// START
public static void alert(){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.notify(); // Alerts a waiting DRT thread in the background.

   }

}
// END
```

A3. This third excerpt is part of the DRT 71, and corresponds to step 128 of Fig. 12. This code fragment shows the DRT in a separate thread, such as thread 121/1 of Fig. 12, after being notified or requested by step 125 and array 127, and sending the changed value and changed value location/identity across the network 53 to the other of the plulality of machines M1...Mn.

```
// START
MulticastSocket ms = DRT.getMulticastSocket();        // The multicast socket
                                                      // used by the DRT for
                                                      // communication.

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");   // Stores
                                                            // the field
                                                            // from the
                                                            // modified
                                                            // class.

// In this example, the field is a byte field.
while (DRT.isRunning()){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.wait();    // The DRT thread is waiting for the alert
                            // method to be called.

      byte[] b = new byte[]{nameTag, field.getByte(null)};  // Stores
                                                            // the
                                                            // nameTag
                                                            // and the
                                                            // value
                                                            // of the
                                                            // field from
                                                            // the modified
                                                            // class in a
                                                            // buffer.

      DatagramPacket dp = new DatagramPacket(b, 0, b.length);

      ms.send(dp);  // Send the buffer out across the network.

   }
}
// END
```

A4. The fourth excerpt is part of the DRT 71, and corresponds to steps 135 and 136 of Fig. 13. This is a fragment of code to receive a propagated identity and value pair sent by another DRT 71 over the network, and write the changed value to the idenitified memory location.

```
// START
MulticastSocket ms = DRT.getMulticastSocket(); // The multicast socket
                                               // used by the DRT for
                                               // communication.

DatagramPacket dp = new DatagramPacket(new byte[2], 0, 2);

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.
```